Amin Sobhani

CS2310 Final Report

Electroencephalogram (EEG) is measure of the electrical activity of the brain. Many practical imaging techniques like nuclear imaging and MRI have been developed over the recent years. However, neurologists still use EEG as a complementary tool for assessment of neurological disorders since EEG records electrical activity of the brain over a period of the time while other imaging techniques do not provide such advantage. EEG has a variety of applications in neurology including diagnosing sleep disorders, infections, tumors.

EEG can be recorded using EEG electrodes. EEG electrodes measure the electric potential difference of the electric field created by brain neural activities, at the scalp. This can be done in two ways. The first way can be performed by choosing an arbitrary referencing electric potential on the head. This is considered as a zero level potential. Having chosen this referencing point, electric voltage of any interest points on the scalp will be calculated by subtracting the recording electrode from the reference electrode. The resulting value shows electrical potential difference of the interest point on the scalp. In the case of multichannel EEG recording, rather than choosing a referencing potential electrode, we can average among different recording electrodes and subtract this average from each electrode.

Researchers use standardized locations for EEG electrodes. This facilitates comparison of different experiment results and studies. Exact locations are different from subject to subject. Figure 1.1 shows the eight standardized locations we used to collect the EEG signals.

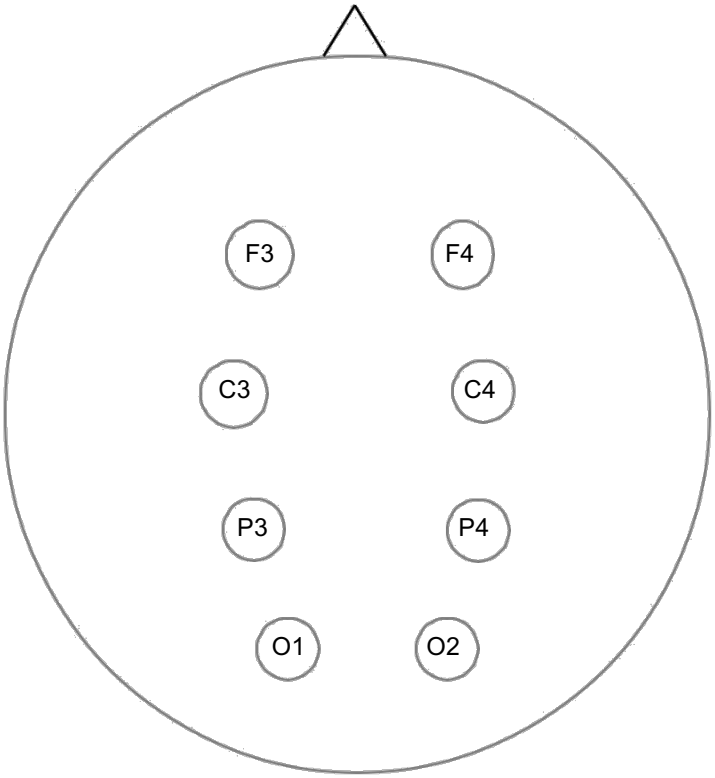


Figure 1.1. Standardized locations used to collect the EEG signals

.

Physicians are not the only ones who use the EEG to diagnose different neurological disorders. Brain Computer Interface (BCI) scientists use the EEG as a way of communication between digital devices and the brain. The EEG is not the only brain function measurement technology. However, researchers widely believe that the EEG is the best technology among all of them. By using the EEG as an alternative way of communication, the BCI provides a significant impact on the assistive care. For every activity, feeling or any decisions we make, there are associated neural activities in our brain. Different neural activities result in different EEG signals. Discriminating these different EEG signals provides unique commands from the brain to the external world. For example, someday a completely paralyzed patient may be able to control robotic arm movement by just thinking about it. BCI applications are not limited to assistive care. BCI systems can also be applied to control a computer application such as word processor or Graphic User Interface. Nowadays, gaming industries use BCI technologies. However, it's in an early stage.

1.2. EEG Paradigm

One of the most important EEG paradigms that has been explored in the BCI systems is the P300 signal. The P300 wave is an endogenous event-related-potential which can be captured during the process of decision making as a subject reacts to a stimulus. A usual way to detect the P300 signal is to show a subject two types of the events occurring at different rates. The event occurring less frequently than the other elicits a positive signal component with a latency of roughly 250-500 ms.

The BCI lab at Colorado State University detected and recorded the elicitation of the P300 signals by asking subjects to look at a computer screen in which different letters randomly appeared in the middle. Then the subjects were asked to count how many times

a specific letter (target letter) was shown in the middle of the screen. The Figure 1.2 shows P300 and nonP300 signals recorded by the BCI lab at Colorado State University.

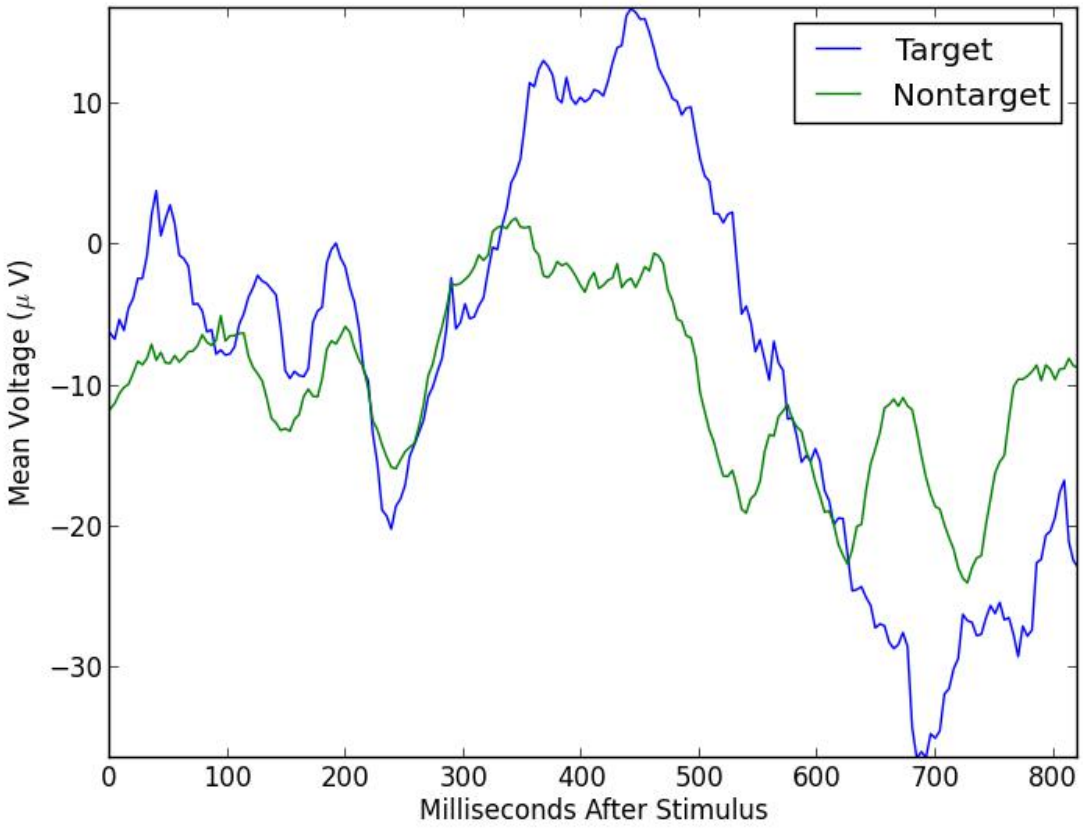


Figure 1.2. The P300 signal

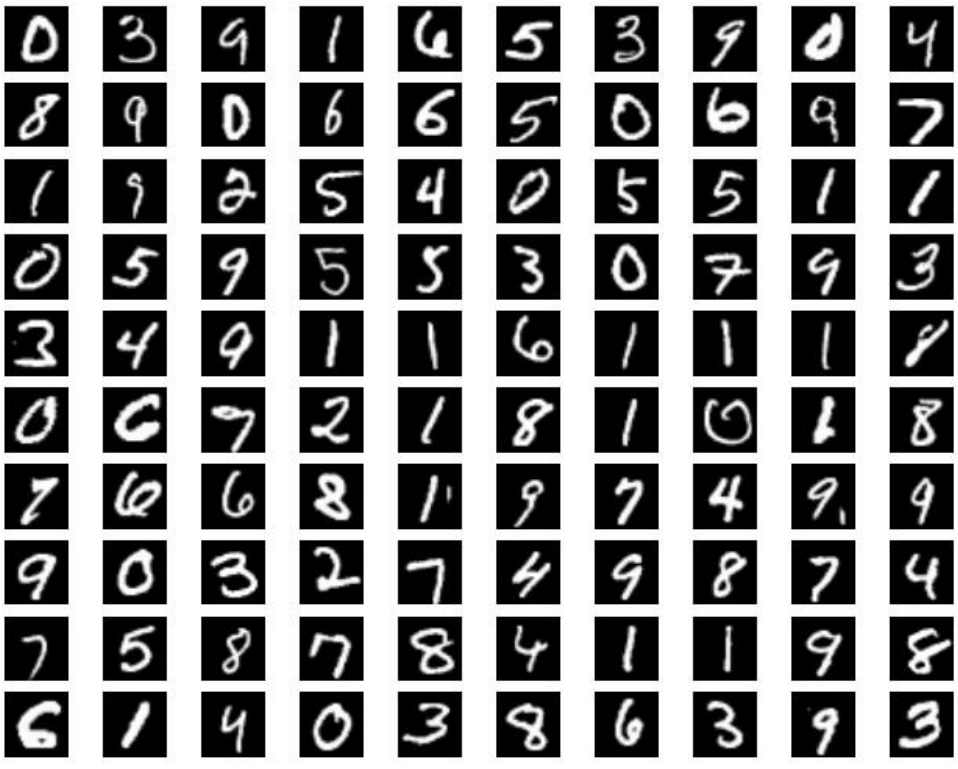
P300 detection has many applications in the BCI field. One of the most common applications of P300 detection is the P300 speller which enables users to type letters on the screen. In the mid 1970s, P300 detection was also used for lie detection.

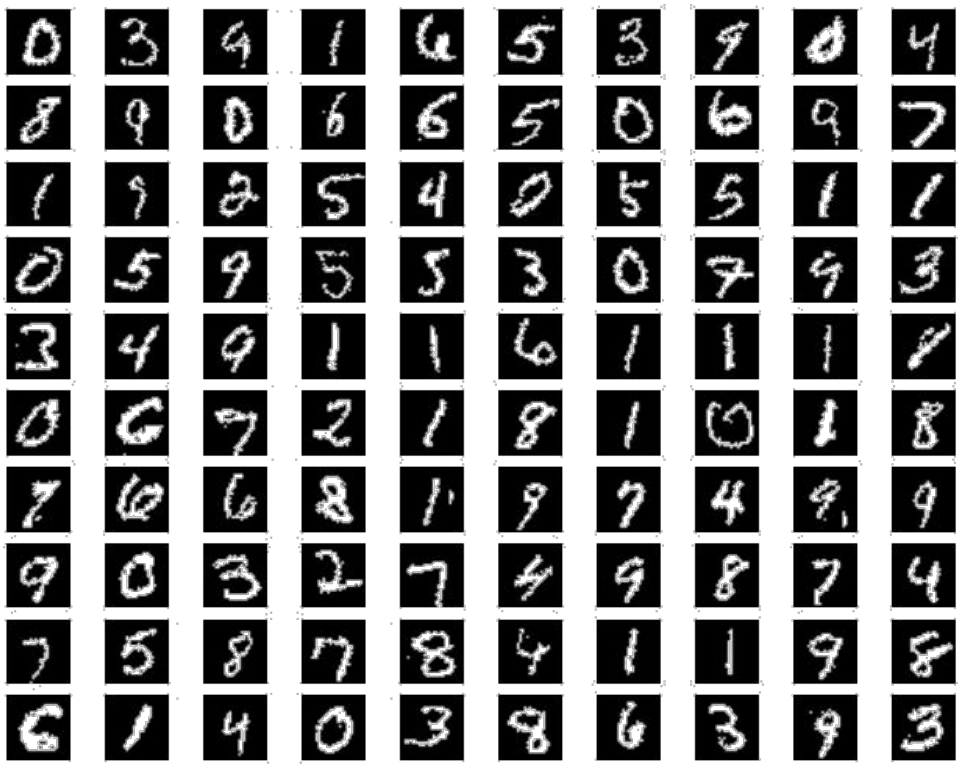
Machine Learning algorithms play a crucial role in designing a BCI system for a variety of reasons. First of all, the machine learning algorithms enable a BCI device to extract useful features for discrimination. The EEG recorded at the scalp is usually noisy for a variety of reasons. The feature extraction module makes it easier for the classifier to discriminate

the EEG signals into its corresponding brain activity pattern. In fact the feature extraction module tends to remove the noise, artifacts and other unnecessary features of the raw data. The second purpose of using the machine learning algorithm in BCI systems is the classification of EEG signals. In order to translate EEG signals to a control signal, BCI systems should first capture the pattern of EEG signals and discriminate them into different command categories. This is usually done using different machine learning-based classifier.

Retracted Boltzmann Machine Training:

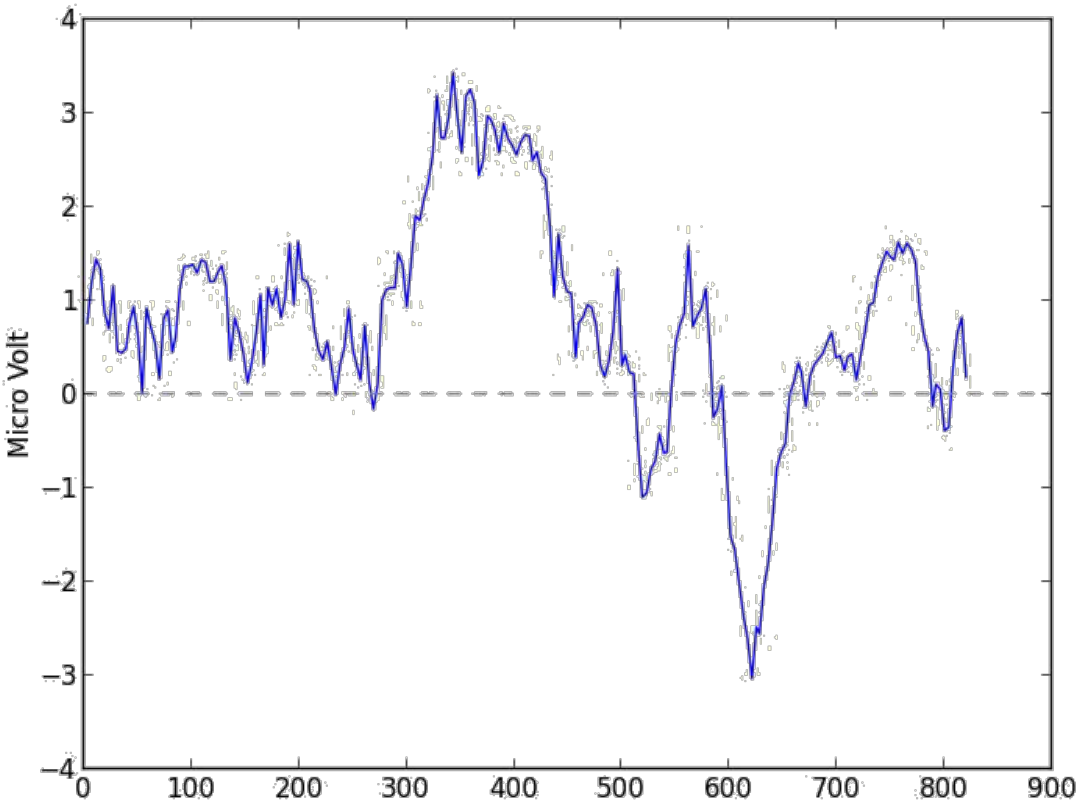
For the sake of this project, I implemented Restricted Boltzmann Machine. Below, I have provided the MNIST data created by Restricted Boltzmann Machine In order to test my implementation. The second set of the image is the reconstructed version of the first set of the digits.

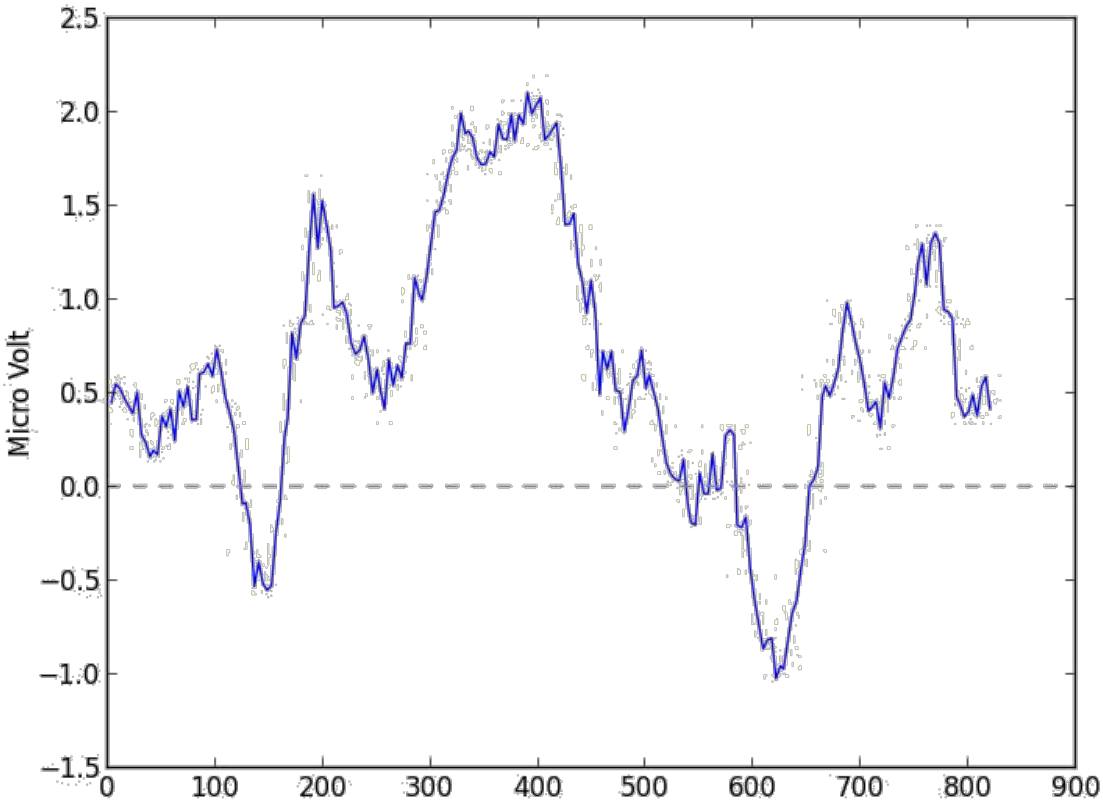




P300 Signal and Recreation of the P300 signal using Restricted Boltzmann Machine

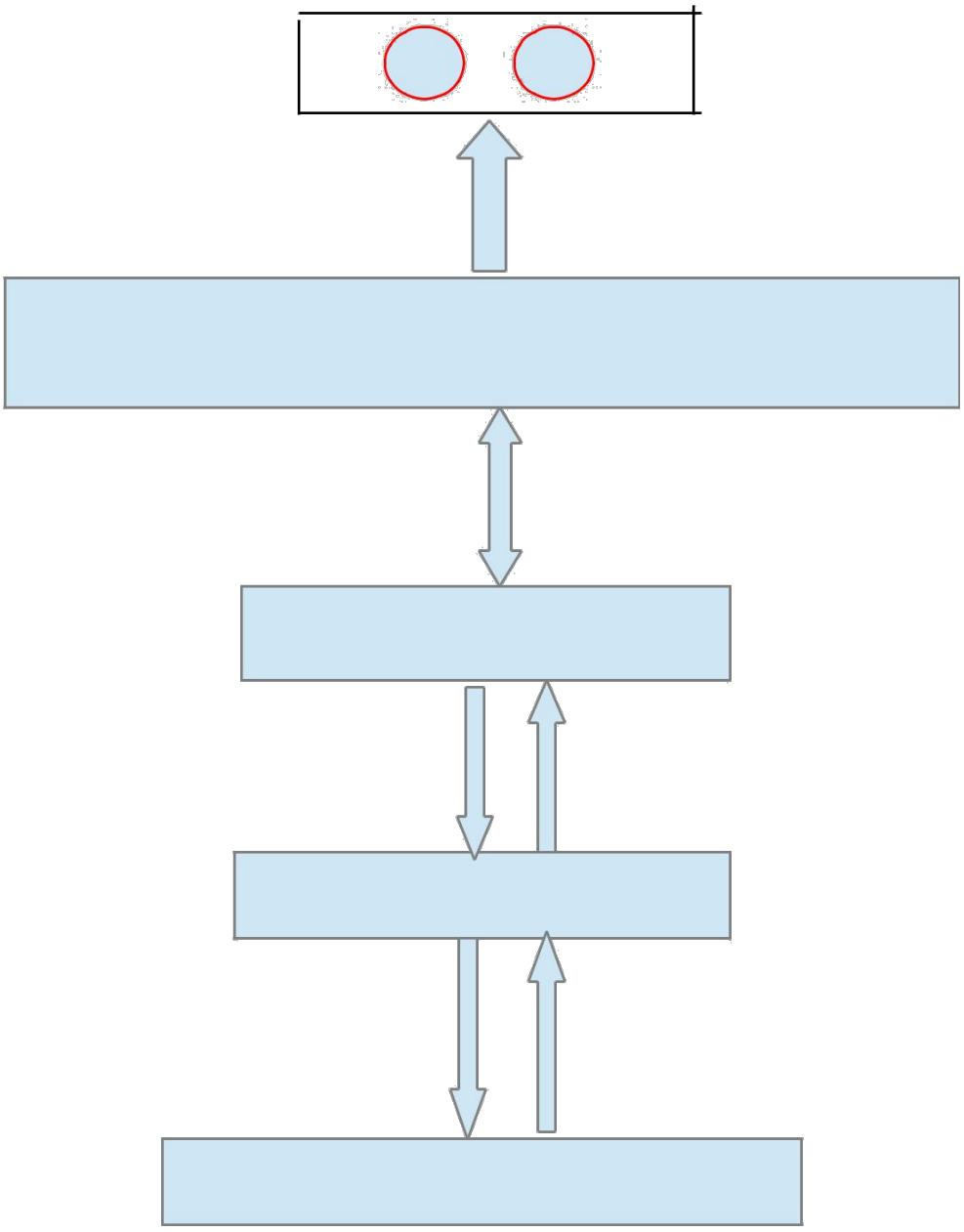
After implementing the restricted Boltzmann machine, I tried to recreate the P300 signal using my RBM. Below, I have provided the P300 signal and the one recreated by RBM.


Micro Volt
4
3
2
1
0
−1
−2
−3
−4
0
100
200
300
400
500
600
700
800
900


Micro Volt
2.5
2.0
1.5
1.0
0.5
0.0
−0.5
−1.0
−1.5
0
100
200
300
400
500
600
700
800
900

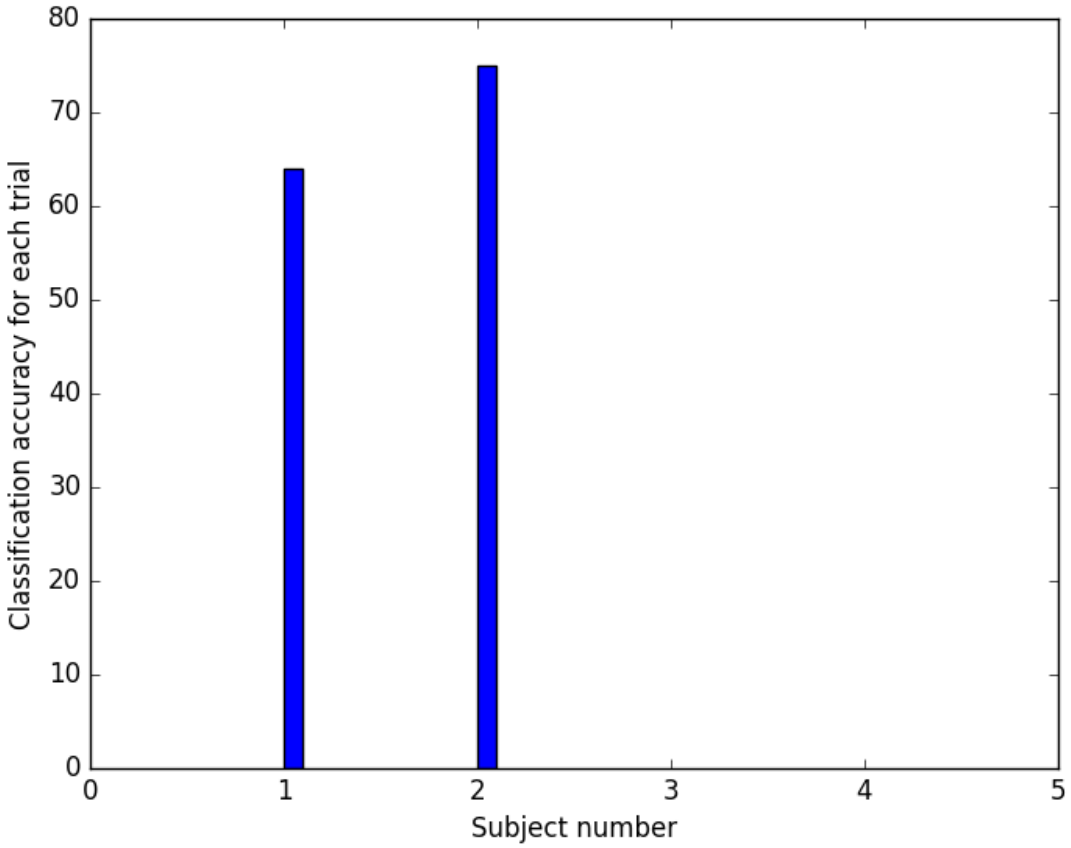
Deep Belief Network Architecture:

Below, I have provided the architecture of my software- The fourth layer has 500 units and the second and the third layer has 150 units. I chose the input dimension to be 150.



Experiments and Results

I used the data provided by Professor Chang in order to conduct some experiments using the machine I developed. I changed the format of the data to .CSV. The Below graph shows the classification accuracy I achieved for two subjects.



Conclusion

I did not have enough training data sets to train my system to reach higher classification accuracy. In order to increase the classification accuracy we achieved, we need more data subjects.

Extra Deeds:

1- Data Format conversion to CSV
2- Conducted Experiment using the new data sets that Dr. Chang provided
3- Conducted RBM experiment using MNIST data set

```
def randNum(a , b):
    return int (round (a * b))
'''
def makePartitions(X,T): # This function partiotions the data so that 60% for traing data and 20% for
validation data set and 20% for testing data set
   nrow= X.shape[0]
   allIndex=xrange(nrow)                                  # lst of all indexes
   nTrain = int(round(nrow*(0.8)))                        # number of the training samles
   #nValidate = int(round(nrow*(0.8)))- int(round(nrow*(0.6)))          # number of the validation samples
   nTest = nrow-nTrain
   trainIndex = list(set(random.sample(allIndex,nTrain)))
   testIndex = list(set(allIndex).difference(set(trainIndex)))     # All indexes minus traing indexes
   #validateIndex = list(set(random.sample(remainingindex,nValidate)))
   #testIndex = list(set(remainingindex).difference(set(validateIndex)))
   Xtrain = X[trainIndex,:]
   Ttrain = T[trainIndex,:]
   Xtest = X[testIndex,:]
   Ttest = T[testIndex,:]
   #Xvalidate = X[validateIndex,:]
   #Tvalidate = T[validateIndex,:]
   return (Xtrain, Ttrain, Xtest, Ttest)
'''

def makePartitions(X,T): # This function partiotions the data so that 60% for traing data and 20% for
validation data set and 20% for testing data set
   nrow= X.shape[0]
   allIndex=range(nrow)                                  # lst of all indexes
   nTrain = int(round(nrow*(0.7)))                        # number of the training samles
   nValidate = int(round(nrow*(0.15)))          # number of the validation samples
   nTest = nrow-nTrain-nValidate
   trainIndex = list(set(random.sample(allIndex,nTrain)))
   remainingindex = list(set(allIndex).difference(set(trainIndex)))     # All indexes minus traing indexes
   validateIndex = list(set(random.sample(remainingindex,nValidate)))
   testIndex = list(set(remainingindex).difference(set(validateIndex)))
   Xtrain = X[trainIndex,:]
   Ttrain = T[trainIndex,:]
   Xtest = X[testIndex,:]
   Ttest = T[testIndex,:]
   Xvalidate = X[validateIndex,:]
```

```
    Tvalidate = T[validateIndex,:]
    return (Xtrain, Ttrain, Xtest, Ttest, Xvalidate, Tvalidate)


def element_wise(m , n , p):
        first=m.shape[0]
        second=m.shape[1]
        return  numpy.asarray([m*n*p for m,n,p in zip(m.flat,n.flat, p.flat)]).reshape(first ,second)


class DBN:


    def __init__(self , ni= 784 , Hidden_Layers = [] , no = 10):

        self.activation = []
        self.ni = ni
        self.no = no
        self.Ws= self.W = numpy.zeros((self.ni , self.no))
        self.bs = numpy.ones((1, self.no))
        self.batch_size = 10
        if not Hidden_Layers:
          self.n_Hidden_layers = 0
        else:
          self.n_Hidden_layers = len(Hidden_Layers)
        self.all_Layers_no = Hidden_Layers
        self.all_Layers_no.append(self.no)
        self.Layers=[]
        for i in range(self.n_Hidden_layers):
            if i ==0:
              self.Layers.append(HiddenLayer(self.ni , Hidden_Layers[i]))
            else :
              self.Layers.append(HiddenLayer(Hidden_Layers[i-1] , Hidden_Layers[i]))
        if not Hidden_Layers:
          self.logistic=lg.LogisticRegression(self.ni , self.no)

        else:
          self.logistic = lg.LogisticRegression(self.all_Layers_no[-2] , self.no)
        self.Layers.append(self.logistic)



    def feed_forward(self, input):
        X = input
        self.activation=[]
        for i in range(self.n_Hidden_layers):
            temp=self.Layers[i].output(X)
            self.activation.append(temp)
            X= temp
        self.activation.append(self.logistic.p_y_given_x(X))
```

```
    def Loss(self, X , TI):

        self.feed_forward(X)
        if self.n_Hidden_layers > 0:
          return self.logistic.loss(self.activation[-2],TI)
        else:
          return self.logistic.loss(X,TI)




    def grad(self, X , TI):
        dw = []
        db = []
        self.feed_forward(X)
        Delta = (self.activation[-1]) - TI
        for i in reversed(range(self.n_Hidden_layers + 1)):
            if i > 0:
              dw.append(numpy.dot(self.activation[i-1].T ,Delta))
              db.append (Delta.sum(axis=0).reshape(1, Delta.shape[1]))
              Delta = element_wise(numpy.dot(Delta , self.Layers[i].W.T) , self.activation[i-1] ,(1 -
self.activation[i-1]))
            else :
              dw.append(numpy.dot(X.T , Delta))
              db.append(Delta.sum(axis=0).reshape((1, Delta.shape[1])))
        dw= dw[::-1]
        db = db[::-1]
        return (dw , db)




    def makeIndicatorVars(self,T):
        if T.ndim == 1:
          T = T.reshape((-1,1))
        return (T == numpy.unique(T)).astype(int)



    def update(self,X, TI, l_r):
        (dw , db)=self.grad(X , TI)
        for i in range(self.n_Hidden_layers + 1):
            self.Layers[i].update(l_r , dw[i] , db[i])



    def pre_train(self,X , T , Xtest , Ttest):
```

```
        train_input = X
        #valid_input = Xvalid
        test_input = Xtest
        for i in range(self.n_Hidden_layers):
          print ("Pre training Layer %s ", (i + 1))
          if i == 0:
            temp = DBN(self.ni , [self.all_Layers_no[i]] , self.no)
            temp.train(train_input , T , train_input, T,  test_input , Ttest , nIterations=10)
            self.Layers[0].W = temp.Layers[0].W
            self.Layers[i].b = temp.Layers[0].b
          else :
            #print 'Here'
            temp = DBN(self.all_Layers_no[i-1] , [self.all_Layers_no[i]] ,  self.no )
            temp.train(train_input , T , train_input, T, test_input , Ttest,   nIterations=10)
            self.Layers[i].W = temp.Layers[0].W
            self.Layers[i].b = temp.Layers[0].b

          self.feed_forward(X)
          train_input = self.activation[i]
          #self.feed_forward(Xvalid)
          #valid_input = self.activation[i]
          self.feed_forward(Xtest)
          test_input = self.activation[i]



    def train(self,X,T, Xtest , Ttest ,Xval, Tval, nIterations=10  , pretrain = False):

        print ("Training started")
        sum = 0
        period = 5
        best = numpy.inf
        training_batch_number = round(X.shape[0] / self.batch_size)
        self.feed_forward(X)
        self.Layers[-1].train(self.activation[-2],T,self.activation[-2],Ttest)
        print ('Started Fine_tuning')
        for j in range((nIterations-5)):
          self.feed_forward(X)
          #numpy.random.shuffle(X)

          for i in range(training_batch_number):
            sys.stdout.write("Epoch %d processing batch %d\r" % (j+1, i+1))
            sys.stdout.flush()
            self.update(X[i * self.batch_size: (i+1)* self.batch_size ,: ] , T[i * self.batch_size: (i+1)*
self.batch_size ,:] , 0.001)
          print ("Epoch : ", (j+1))
          sum += self.Loss(Xval, Tval)
          if (j+1) % period == 0:
            new = numpy.float(sum/period)
            sum = 0
```

```
            if new > best:
              break
            if new < best:
              best = new



def use(self, X):
    self.feed_forward(X)
    if self.n_Hidden_layers > 0:
      return  self.Layers[self.n_Hidden_layers].use(self.activation[-2])
    else:
      return self.Layers[0].use(X)

def makeStandardize(self,X):
    means = X.mean(axis=0)
    stds = X.std(axis=0)
    stds[stds == 0 ] = 1.e-20
    def standardize(origX):
        return (origX - means) / stds
    def unStandardize(stdX):
        return stds * stdX + means
    return (standardize, unStandardize)


def percentCorrect(self,p,t):
   return numpy.sum(p.ravel()==t.ravel()) / float(len(t)) * 100


def RBM_Pre_training(self,X , T):

   train_input = X

   for i in range(self.n_Hidden_layers):
      print ("RBM_Pre_training Layer %s ", (i + 1))
      if i == 0:
         temp = VRBM(n_visible = self.ni, n_hidden = self.all_Layers_no[0])
         temp.train(train_input)
         self.Layers[0].W = temp.W
         self.Layers[i].b = temp.hbias
      else :
         temp = RBM(n_visible = self.all_Layers_no[i-1] , n_hidden = self.all_Layers_no[i])
         temp.train(train_input)
         self.Layers[i].W = temp.W
         self.Layers[i].b = temp.hbias

      (active , sig ) = temp.vis_prop(train_input)
      train_input = sig
```

```
def randNum(a , b):
    return int (round (a * b))

class LogisticRegression(object):

    def __init__(self,ni,no):

        self.ni=ni
        self.no = no
        self.W = numpy.zeros((self.ni , self.no))
        self.b = numpy.ones((1, self.no))
        self.batch_size = 20

    def use(self, X):
        return numpy.argmax(self.p_y_given_x(X) , axis=1)

    def weight_update(self, WN):
        WN = numpy.array(WN)
        self.W = WN

    def makeIndicatorVars(T):
        if T.ndim == 1:
            T = T.reshape((-1,1))
        return (T == numpy.unique(T)).astype(int)

    def softmax(self,w):
        w = numpy.array(w)
        maxes = numpy.amax(w, axis=1)
        maxes = maxes.reshape(maxes.shape[0], 1)
        e = numpy.exp(w-maxes)
        #kir =  numpy.sum(e, axis=1).reshape(e.shape[0],1)
        #print kir
        #print kir[numpy.where(kir > 0)]

        dist = e / numpy.sum(e, axis=1).reshape(e.shape[0],1)
        return dist

    def p_y_given_x(self,X):

        dt = numpy.dot(X, self.W) + self.b
        return self.softmax(dt)

    def gradient(self,X , T ):
        delta =self.p_y_given_x(X) - T
        dW = numpy.dot(X.T , delta)
        db = delta.mean(axis=0).reshape(self.b.shape[0], self.b.shape[1])
        return (dW, db)

    def loss(self, input , target):
```

```
        m = numpy.log((self.p_y_given_x(input)))
        n = target
        return -numpy.exp(numpy.sum([m*n for m,n in zip(m.flat, n.flat)])/input.shape[0])



    def update(self,learning_rate , dW, db):
        self.W = self.W - learning_rate * dW
        self.b = self.b - learning_rate * db

    def percentCorrect(self,p,t):
        return numpy.sum(p.ravel()==t.ravel()) / float(len(t)) * 100

    def One_step(self, data , T , l_r):
        (dW , db) = self.gradient(data , T)
        self.update(l_r , dW , db)
        #print self.loss(data, T)


    def train(self, X, T , VX , VT , n_epochs = 5):

        print ('Training started')
        training_batch_number = X.shape[0]
        epoch = 0
        best_score=numpy.inf
        while (epoch < n_epochs):
          epoch = epoch + 1
          numpy.random.shuffle(X)
          for i in range(0,training_batch_number):
              self.One_step(X[i*self.batch_size:(i+1)*self.batch_size,:] ,
T[i*self.batch_size:(i+1)*self.batch_size,:], 0.0001)

          if (epoch) % 10 == 0:
            this_score = self.loss(VX , VT)
            if  this_score < best_score :
                best_score = this_score
          print('epoch %i, training_likelihood %f %%, validation_likelihood %f %% ,
training_classification_accuracy %f %% , validation_classification_accuracy %f' %(epoch , self.loss(X,T)
, self.loss(VX,VT) , self.percentCorrect(self.use(X) , numpy.argmax(T,axis=1)) ,
self.percentCorrect(self.use(VX) , numpy.argmax(VT,axis=1))))


    def makeIndicatorVars(self,T):
        if T.ndim == 1:
          T = T.reshape((-1,1))
        return (T == numpy.unique(T)).astype(int)

    def makeStandardizeF(self,X):
        means = X.mean(axis=0)
        stds = X.std(axis=0)
        stds[stds == 0 ] = 1.e-100
```

```
        def standardize(origX):
            return (origX - means) / stds
        def unStandardize(stdX):
            return stds * stdX + means
        return (standardize, unStandardize)


if __name__ == '__main__':

    print ('Started loading data')

    fname_lbl = 'train-labels-idx1-ubyte'
    flbl = open(fname_lbl, 'rb')
    magic_nr, size = struct.unpack(">II", flbl.read(8))
    lbl = array("b", flbl.read())
    flbl.close()

    fname_img = 'train-images.idx3-ubyte'
    fimg = open(fname_img, 'rb')
    magic_nr, size, rows, cols = struct.unpack(">IIII", fimg.read(16))
    img = array("B", fimg.read())
    fimg.close()

    test_fname_lbl = 't10k-labels-idx1-ubyte'
    test_flbl = open(test_fname_lbl, 'rb')
    magic_nr, test_size = struct.unpack(">II", test_flbl.read(8))
    test_lbl = array("b", test_flbl.read())
    test_flbl.close()

    test_fname_img = 't10k-images-idx3-ubyte'
    test_fimg = open(test_fname_img, 'rb')
    test_magic_nr, test_size, test_rows, test_cols = struct.unpack(">IIII", test_fimg.read(16))
    test_img = array("B", test_fimg.read())
    test_fimg.close()


    ind = [ k for k in xrange(size)]
    tr_images =  numpy.zeros(( len(ind), rows , cols))
    tr_images_flat = numpy.zeros((len(ind),rows*cols))
    tr_labels = numpy.zeros((len(ind), 1))
    test_ind = [ k for k in xrange(test_size)]
    test_images =  numpy.zeros(( len(test_ind), test_rows , test_cols))
    test_labels = numpy.zeros((len(test_ind), 1))
    for i in xrange(len(ind)):
        tr_images[i] = numpy.array(img[ ind[i]*rows*cols : (ind[i]+1)*rows*cols ]).reshape((rows, cols))
        tr_images_flat[i] = tr_images[i].flatten()
        tr_labels[i] = lbl[ind[i]]

    for i in xrange(len(test_ind)):
        test_images[i] = numpy.array(test_img[ test_ind[i]*test_rows*test_cols :
(test_ind[i]+1)*test_rows*test_cols ]).reshape((test_rows, test_cols))
```

```
        test_labels[i] = test_lbl[test_ind[i]]

    nrow=X.shape[0]
    nTrain = randNum( nrow , float(0.1))
    nValid = nrow - nTrain
    rowIndicesRandomlyOrdered = random.sample(xrange(nrow),nrow)
    trainIndices = rowIndicesRandomlyOrdered[:nTrain]
    validIndices = rowIndicesRandomlyOrdered[nTrain:]
    Xtrain = X[trainIndices,:]
    Ttrain = T[trainIndices,:]
    Xvalid = X[validIndices,:]
    Tvalid = T[validIndices,:]

################################################################################################
####


    print ('Reading Parkinson Data')
    f = open("parkinsons.data","r")
    header = f.readline()
    names = header.strip().split(',')[1:]


    data = numpy.loadtxt(f ,delimiter=',', usecols=1+numpy.arange(23))

    targetColumn = names.index("status")
    XColumns = range(23)
    XColumns.remove(targetColumn)
    X = data[:,XColumns]
    T = data[:,targetColumn,numpy.newaxis]
    names.remove("status")

    trainf = 0.8
    healthyI,_ = numpy.where(T == 0)
    parkI,_ = numpy.where(T == 1)
    healthyI = numpy.random.permutation(healthyI)
    parkI = numpy.random.permutation(parkI)

    nHealthy = len(healthyI)
    nParks = len(parkI)

    n = round(trainf*len(healthyI))
    rows = healthyI[:n]
    Xtrain = X[rows,:]
    Ttrain = T[rows,:]
    rows = healthyI[n:]
    Xtest =  X[rows,:]
    Ttest =  T[rows,:]
    n = round(trainf*len(parkI))
    rows = parkI[:n]
    Xtrain = numpy.vstack((Xtrain, X[rows,:]))
```

```
Ttrain = numpy.vstack((Ttrain, T[rows,:]))
rows = parkI[n:]
Xtest = numpy.vstack((Xtest, X[rows,:]))
Ttest = numpy.vstack((Ttest, T[rows,:]))



class RBM:


  def __init__(self, n_visible, n_hidden, W =None , hbias = None , vbias = None , learning_rate =
0.1):

      self.n_visible = n_visible
      self.n_hidden = n_hidden
      self.learning_rate= learning_rate
      if W == None:
        #self.W = numpy.random.uniform(-4 * numpy.sqrt(6. / (self.n_visible + self.n_hidden)) , 4 *
numpy.sqrt(6. / (self.n_visible + self.n_hidden)) , size = (self.n_visible , self.n_hidden))
        self.W=numpy.random.normal(0.0, 0.1 , (self.n_visible , self.n_hidden))
        #self.W = numpy.zeros((self.n_visible , self.n_hidden))
      else:
        self.W = W

      if hbias == None:
         self.hbias = numpy.zeros((1 , self.n_hidden))
      else :
         self.hbias = hbias


      if vbias == None:
         self.vbias = numpy.zeros((1 , self.n_visible))
      else :
         vbias = vbias
      self.batch_size = 10
      self.weightUpdates = numpy.zeros((self.n_visible , self.n_hidden))
      self.hbiasUpdates  = numpy.zeros((1 , self.n_hidden))
      self.vbiasUpdates = numpy.zeros((1 , self.n_visible))
      self.hB = self.hbias
      self.vB = self.vbias
  def sigmoid(self, x):
       return 1.0 / (1 + numpy.exp(-x))

  def vis_prop(self, data):

      temp = numpy.dot (data , self.W ) + self.hbias
      return (temp, self.sigmoid(temp))

  def hid_prop(self, data):
      activation = numpy.dot(data, self.W.T) + self.vbias
```

```
    sig = self.sigmoid(activation)
    return (activation , sig)

def stochastic_binary_sample_h(self, data):

     hidden_states = (data > numpy.random.rand(data.shape[0], self.n_hidden))
     return hidden_states.astype(numpy.float)


def stochastic_binary_sample_v(self, data):

     visible_states=(data > numpy.random.rand(data.shape[0], self.n_visible))
     return visible_states.astype(numpy.float)

def gausian_sample_v(self, data):
     visible_states= data + numpy.random.randn(data.shape[0] ,self.n_visible )
     return visible_states.astype(numpy.float)


def gibbs_hvh(self, data):

     output1 = self.stochastic_binary_sample_v(data)
     output2 = self.stochastic_binary_sample_h(output1)
     return output2

def gibbs_vhv(self, data):

     (hid_act , hid_sig) = self.vis_prop(data)
     output1 = self.stochastic_binary_sample_h(hid_sig)
     (vis_act , vis_sig) = self.hid_prop(output1)
     output2 = self.stochastic_binary_sample_v(vis_sig)
     return (hid_act , hid_sig , output1 , vis_act , vis_sig , output2)
def err_density(self, base_val , batchval):
    return base_val + batchval

def makeStandardize(self,X):
    means = X.mean(axis=0)
    stds = X.std(axis=0)
    stds[stds == 0 ] = 1.e-200
    def standardize(origX):
       return (origX - means) / stds
    def unStandardize(stdX):
       return stds * stdX + means
    return (standardize, unStandardize)


def train(self, TX, max_epochs = 200):
  #momentum = [0.5*ones(1,5) 0.9*ones(1,1000-5)];
  first = [0.5 for i in range(5)]
  second = [0.9 for i in range(max_epochs - 5)]
  momentum = first + second
```

```
    weightCost = [0.0002 for i in range(max_epochs)]
    training_batch_number = TX.shape[0] / self.batch_size
    Training_Data_Number = TX.shape[0]
    t_error_start = numpy.inf
    v_error_start = numpy.inf
    stop = False

    for iteration in range(max_epochs):
      #numpy.random.shuffle(TX)
      #self.plot_progress(1)
      #numpy.random.shuffle(TX)
      base_val = 0
      if not stop :
        for i in range(10):
          sys.stdout.write("Epoch %d processing batch %d\r" % (iteration+1, i+1))
          sys.stdout.flush()
          #self.plot_progress(2)
          current_batch = TX[i*self.batch_size:(i+1)*self.batch_size,:]
          (active , sig ) = self.vis_prop(current_batch)
          binary_sample = self.stochastic_binary_sample_h(sig)
          pos_associations = numpy.dot(current_batch.T, sig)
          (vis_act, vis_sig) = self.hid_prop(binary_sample)
          (hid_act , hid_sig) = self.vis_prop(vis_sig)
          #(hid_act , hid_sig) = self.vis_prop(vis_act)
          neg_associations = numpy.dot(vis_sig.T, hid_sig)
          #neg_associations = numpy.dot(vis_act.T, hid_sig)
          self.weightUpdates = (self.learning_rate *\
          (((pos_associations - neg_associations) / self.batch_size ) - \
          0.0001 * self.W)) + (momentum[iteration] * self.weightUpdates)
          self.hbiasUpdates = self.learning_rate * \
          ((numpy.mean(sig) - numpy.mean(hid_sig) - 0.0001 * \
          self.hbias)) + (momentum[iteration] * self.hbiasUpdates)
          self.vbiasUpdates = self.learning_rate * \
          (numpy.mean(current_batch) - numpy.mean(vis_sig) - 0.0001 * \
          self.vbias) + (momentum[iteration] * self.vbiasUpdates)
          #self.W += (self.learning_rate * (((pos_associations - neg_associations) / self.batch_size ) -
0.0002 * self.W))
          self.W += self.weightUpdates
          self.hbias += self.hbiasUpdates
          self.vbias += self.vbiasUpdates
          (o1,o2,o3,o4,o5,o6) = self.gibbs_vhv(current_batch)
##################################
          batch_error = numpy.sum((current_batch - o5) ** 2)
##################################
          #batch_error = numpy.sum((current_batch - vis_act) ** 2)
          base_val = self.err_density(base_val , batch_error)
####################################
          #self.hbias += self.learning_rate * (numpy.mean(sig) - numpy.mean(hid_sig))
          #self.vbias += self.learning_rate * (numpy.mean(current_batch) - numpy.mean(vis_sig))
####################################
      (o1,o2,o3,o4,o5,o6) = self.gibbs_vhv(TX)
```

```
    T_error = numpy.sum((TX - o5) ** 2)
######################
    #print "Epoch %s: Training_Error_Value is %s" % (iteration+1, T_error)
########################
    print ("Epoch %s: Training_Error_Value is %s" ,(iteration+1,base_val))
    if iteration == 0 :
      t_error_start = T_error
    #if ((iteration > 30) and ((T_error > t_error_start) or (V_error > v_error_start))):
      #break
```