Warsaw University of Technology

Faculty of mathematics and
Information Sciences

First semester 2021/2022

Neural network

Project 3

By

Swetha Suresh Kumar (324353)

Andra Umoru (324334)

Submitted To:

Stanislaw Kazmierczak

Table of Contents

List of Figures

1. Problem description

Implementation of the Adaptive Resonance Theory for MNIST dataset.

- Compare different vigilance values
- Compare clusters with MNIST classes.

2. Theoretical Description

2.1 Adaptive Resonance Theory (ART)

Adaptive Resonance Theory (ART) is a cognitive and neurological theory that explains how the brain learns to attention to, categorize, recognize, and predict things and events in a changing environment on its own. ART presently has the most comprehensive set of cognitive and neurological theories for explanation and prediction. The ability of ART to autonomously carry out quick, incremental, unsupervised and supervised learning in response to a changing reality, without destroying previously learned memories, is essential to its predictive capability.

The stability-plasticity issue of a system is addressed by the Adaptive Resonance Theory, which asks how learning may progress in response to large input patterns while maintaining stability for irrelevant patterns. Aside from that, the stability-elasticity paradox is concerned with how a system can adapt to new data while maintaining previous knowledge. A feedback mechanism is added among the ART neural network layers for such a task. The data in the form of processing element output reflects back and forth across layers in this neural network. Adaption can occur during this period if an adequate pattern is built up and the resonance is reached.

Stability: The stability of the ART architecture connotes those irrelevant events have no effect on system behavior.

Plasticity: The system adapts its behavior in response to important occurrences.

Dilemma: The dilemma asks the following:

- How to achieve stability while avoiding rigidity and chaos.
- Continuous learning ability.
- Learning knowledge is preserved.

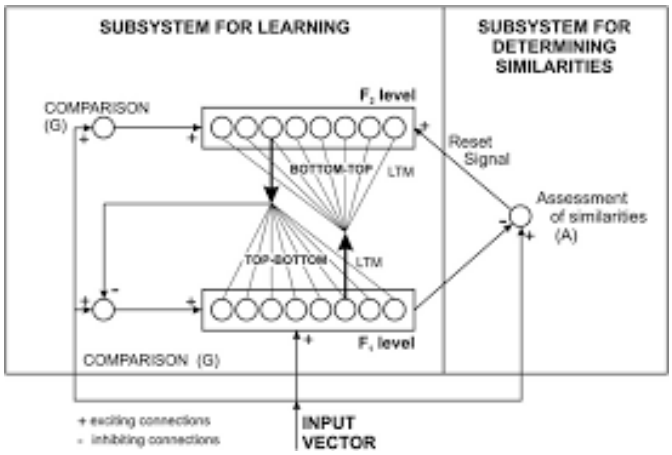


Figure 1: ART for binary Input

2.2 Advantage of Adaptive Resonance Theory (ART)

i. It is stable and unaffected by a wide range of inputs to its network.
ii. It can be combined with a variety of different approaches to produce even better outcomes.
iii. It can be utilized in a variety of applications, including mobile robot control, face identification, land cover classification, target recognition, medical diagnosis, signature verification, and web user clustering, among others.
iv. It offers benefits over competitive learning. Competitive learning does not have the capacity to add additional clusters as needed.
v. It does not ensure cluster formation stability.

3. Algorithm for ART

The figure below as captured from the class lectures slides shows how the ART problem is solved:

- Initialize each top-down weight $t_{l,j}(0) = 1$;
- Initialize each bottom-up weight $b_{j,l}(0) = \dfrac{1}{n+1}$;
- **while** the network has not stabilized, **do**

1. Present a randomly chosen pattern $x = (x_1, \ldots, x_n)$ for learning.
2. Let the active set A contain all nodes; calculate $y_j = b_{j,l}x_1 + \ldots + b_{j,n}x_n$ for each node $j \in A$;

   (a) Let j^* be a node in A with largest y_j, with ties being broken arbitrarily;

   (b) Compute $s^* = (s_1^*, \ldots, s_n^*)$ where $s_l^* = t_{l,j^*}x_l$;

   (c) Compare similarity between s^* and x with the given vigilance parameter ρ:

   if $\dfrac{\sum_{l=1}^{n} s_l^*}{\sum_{l=1}^{n} x_l} \leq \rho$ then remove j^* from set A

   else associate x with node j^* and update weights:

$$b_{j^*,l}\,(\text{new}) = \frac{t_{l,j^*}\,(\text{old})\,x_l}{0.5 + \sum_{l=1}^{n} t_{l,j^*}\,(\text{old})\,x_l}$$

$$t_{l,j^*}\,(\text{new}) = t_{l,j^*}\,(\text{old})\,x_l$$

   until A is empty or x has been associated with some node j;

3. If A is empty, then create a new node whose weight vector coincides with the current input pattern x;

end-while

Figure 2: Algorithm for ART Architecture

4. Implementation

This implementation was carried out using python programming language. The dataset used for this implementation is MNIST dataset. This dataset was converted from the image dataset to binary form. The implementation was carried using the algorithm above.

4.1. Requirements

2.1 Requirements

The application is implemented using python 3. The libraries used are:

1. Numpy
2. Pandas
3. Matplotlib.pyplot
4. Random
5. Keras.datasets
6. Sklearn.metrics.cluster

4.2. Implementation of the algorithm

4.2.1.Data preparation and division into training and testing dataset

We have converted the MNIST dataset image to binary. And divided it into training and testing dataset (60.000 / 10.000 respectively)

```
[6]  def grey_to_bin(data):
       dataset = []
       for d in data:
         array = []
         for i in d:
           row = []
           for j in i:
             if j != 0:
               row.append(1)
             else:
               row.append(0)
           array.append(row)
         dataset.append(array)
       return np.array(dataset)

     def dimension_reduction(data):
       m, n, k = data.shape
       new_data = []
       for i in range(m):
         new_data.append(data[i].reshape(n*k,))
       return np.array(new_data)

     (train_X, train_y), (test_X, test_y) = mnist.load_data()
     print('X_train: ' + str(train_X.shape))
     print('Y_train: ' + str(train_y.shape))
     print('X_test:  ' + str(test_X.shape))
     print('Y_test:  ' + str(test_y.shape))

     train_X = grey_to_bin(train_X)
     test_X = grey_to_bin(test_X)

     train_X_low_dim = dimension_reduction(train_X)
     test_X_low_dim = dimension_reduction(test_X)
```

Figure 3: Conversation of image to binary

4.2.2. Primary functions

- compare_value: This function is used to calculate a value which is used to compare with vigilance value
- vigil_cond: We used this function to check whether the vigilance is satisfied or not, which will return true if satisfied and false if not satisfied
- update_B: It is used to update the bottom-up weight
- update_T: Used to update the top-down weight

```
def init_params(input_vectors):
    n, m = input_vectors.shape

    T = np.ones((1, m), dtype=float)
    B = np.empty((1, m), dtype=float)
    B.fill(1/(m + 1))

    default_T = np.ones((1, m), dtype=float)
    default_B = np.empty((1, m), dtype=float)
    default_B.fill(1/(m + 1))

    previous_T = np.zeros((1, m), dtype=float)
    previous_B = np.zeros((1, m), dtype=float)

    return T, B, default_T, default_B, previous_T, previous_B

def is_stable(previous_matrix, current_matrix):
  return np.array_equal(previous_matrix, current_matrix)


def calculate_y(input_vector, B):
  #m, n = B.shape
  y = []
  for i in B:
    tmp = np.sum(i * input_vector)
    y.append(tmp)

  y = np.array(y)
  return y

def max_val_index(array, exeption):
  max = 0
  index = 0
  for i in range(len(array)):
    if array[i] not in exeption:
      if array[i] > max:
        max = array[i]
        index = i
  return max, index
```

Figure 4: Primary functions

```
def compare_value(input_vector, T, index_of_T):
  numerator = np.sum(input_vector * T[index_of_T])
  denominator = np.sum(input_vector)
  return numerator / denominator

def vigil_cond(calculated_value, vigilance_value):
  if calculated_value > vigilance_value:
    return True
  return False

def condition_checker(vector, b, vig_value, T):
  y_array = calculate_y(vector, b)
  exeption = []
  vigilance_satisfied = False
  while y_array.shape[0] != len(exeption):
    value, index = max_val_index(y_array, exeption)
    if vigil_cond(compare_value(vector, T, index), vig_value):
      vigilance_satisfied = True
      break
    else:
      exeption.append(value)
  return index, value, y_array, vigilance_satisfied


def update_B(input_vector, T, index, B):
  numerator = T[index] * input_vector
  denominator = 0.5 + np.sum(numerator)
  value = numerator / denominator
  tmp = np.copy(B)
  tmp[index] = value
  return tmp

def update_T(input_vector, T, index, B):
  value = T[index] * input_vector
  tmp = np.copy(T)
  tmp[index] = value
  return tmp

def generate_new_B(input_vector, T, default_T, B, default_B):
  new_B = update_B(input_vector, default_T, 0, default_B)
  return np.vstack((B, new_B))

def generate_new_T(input_vector, T, default_T, B, default_B):
  new_T = update_T(input_vector, default_T, 0, default_B)
  return np.vstack((T, new_T))
```

Figure 5: primary functions

4.2.3.Testing

We have used the following input vectors for testing purposes.

1. (1,1,0,0,0,0,1)
2. (0,0,1,1,1,1,0)
3. (1,0,1,1,1,1,0)
4. (0,0,0,1,1,1,0)

5. (1,1,0,1,1,1,0)

- Vigilance parameter – 0.7

```
#TESTING PURPOSE
v = []
v.append([1, 1, 0, 0, 0, 0, 1])
v.append([0, 0, 1, 1, 1, 1, 0])
v.append([1, 0, 1, 1, 1, 1, 0])
v.append([0, 0, 0, 1, 1, 1, 0])
v.append([1, 1, 0, 1, 1, 1, 0])
input_vectors = np.array(v)
p = 0.7

t = np.ones((1, 7), dtype=float)
b = np.empty((1, 7), dtype=float)
b.fill(1/(7 + 1))

default_t = np.copy(t)
default_b = np.copy(b)

print(v)
print(b)
print(t)

prev_T = np.zeros((1, 7), dtype=float)
prev_B = np.zeros((1, 7), dtype=float)
```

Figure 6: Testing

```
[13] def plot_image(dataset, index):
       plt.subplots(nrows=0, ncols=0)
       plt.figure(figsize=(10, 6))
       plt.imshow(dataset[index], cmap=plt.get_cmap('gray'))
       plt.show()
```

Figure 7: plot

5. Results

5.1. Data preparation and division into training and testing dataset

After training and testing the MNIST dataset, the following results were observed.

```
Downloading data from https://storage.googleapis.com/tensorflow/tf-keras-datasets/mnist.npz
11493376/11490434 [==============================] - 0s 0us/step
11501568/11490434 [==============================] - 0s 0us/step
X_train: (60000, 28, 28)
Y_train: (60000,)
X_test:  (10000, 28, 28)
Y_test:  (10000,)
```

Figure 8: Data preparation and division

5.2. Primary functions

```
iteration --> 0
iteration --> 1000
iteration --> 2000
iteration --> 3000
iteration --> 4000
iteration --> 5000
iteration --> 6000
iteration --> 7000
iteration --> 8000
iteration --> 9000
previous_T -> (1, 784)
previous_B -> (1, 784)
updated T  -> (8, 784)
updated B  -> (8, 784)
iteration --> 0
iteration --> 1000
iteration --> 2000
iteration --> 3000
iteration --> 4000
iteration --> 5000
iteration --> 6000
iteration --> 7000
iteration --> 8000
iteration --> 9000
previous_T -> (8, 784)
previous_B -> (8, 784)
updated T  -> (8, 784)
updated B  -> (8, 784)
2
```

Figure 9: Update of top-down weight and bottom-up weight

```
from sklearn.metrics.cluster import contingency_matrix

b = B.copy()
x = train_y[0:10000].copy()
y = []
for vector in train_X_low_dim[0:10000]:
    y_array = calculate_y(vector, b)
    index = y_array.argmax()
    y.append(index)
y = np.array(y)

print(x.shape)
print(y.shape)
result = contingency_matrix(x, y)
print(result)
```

```
(10000,)
(10000,)
[[148   2  35  20   1 747   2  46]
 [ 39 298   1   0   0 285   0 504]
 [ 56  74   2   6  16 705   2 130]
 [ 29  14  36   4   5 874   0  70]
 [196 111  31  17   6  72   1 546]
 [109  18  27   1   2 428   0 278]
 [  5 160  32  22   1 493   6 295]
 [352 154  98 237  22 170   0  37]
 [ 21  11   2   0   0 755   0 155]
 [ 63  76  27  16   1 506   0 289]]
```

Figure 10: Contingency matrix

5.3. Testing

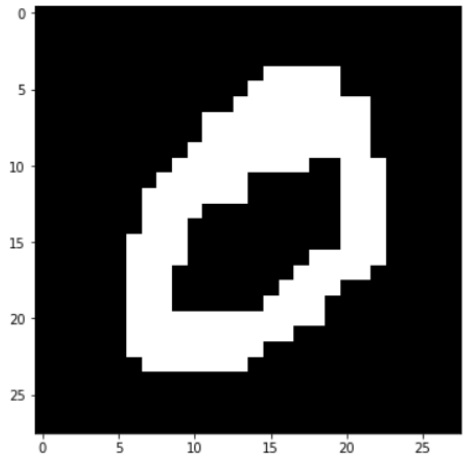
Screenshot showing the expected result.

```
[[1, 1, 0, 0, 0, 0, 1], [0, 0, 1, 1, 1, 1, 0], [1, 0, 1, 1, 1, 1, 0], [0, 0, 0, 1, 1, 1, 0], [1, 1, 0, 1, 1, 1, 0]]
[[0.125 0.125 0.125 0.125 0.125 0.125 0.125]]
[[1. 1. 1. 1. 1. 1. 1.]]
STARTING.....
vector:[1 1 0 0 0 0 1]
array :[0.375]
1.0
[[0.28571429 0.28571429 0.         0.         0.         0.
  0.28571429]]
[[1. 1. 0. 0. 0. 0. 1.]]

Count 0
vector:[0 0 1 1 1 1 0]
array :[0.]
0.0
[[0.28571429 0.28571429 0.         0.         0.         0.
  0.28571429]
 [0.         0.         0.22222222 0.22222222 0.22222222 0.22222222
  0.        ]]
[[1. 1. 0. 0. 0. 0. 1.]
 [0. 0. 1. 1. 1. 1. 0.]]

Count 1
vector:[1 0 1 1 1 1 0]
array :[0.28571429 0.88888889]
0.8
[[0.28571429 0.28571429 0.         0.         0.         0.
  0.28571429]
 [0.         0.         0.22222222 0.22222222 0.22222222 0.22222222
  0.        ]]
[[1. 1. 0. 0. 0. 0. 1.]
 [0. 0. 1. 1. 1. 1. 0.]]

Count 2
vector:[0 0 0 1 1 1 0]
array :[0.         0.66666667]
1.0
[[0.28571429 0.28571429 0.         0.         0.         0.
  0.28571429]
 [0.         0.         0.         0.28571429 0.28571429 0.28571429
  0.        ]]
[[1. 1. 0. 0. 0. 0. 1.]
 [0. 0. 0. 1. 1. 1. 0.]]

Count 3
vector:[1 1 0 1 1 1 0]
array :[0.57142857 0.85714286]
0.4
[[0.28571429 0.28571429 0.         0.         0.         0.
  0.28571429]
 [0.         0.         0.         0.28571429 0.28571429 0.28571429
  0.        ]
 [0.18181818 0.18181818 0.         0.18181818 0.18181818 0.18181818
  0.        ]]
[[1. 1. 0. 0. 0. 0. 1.]
 [0. 0. 0. 1. 1. 1. 0.]
 [1. 1. 0. 1. 1. 1. 0.]]
```

Figure 11: Testing

```
plot_image(train_X, 1)
print(train_y[2])
```

```
<Figure size 432x288 with 0 Axes>
```




```
4
```

Figure 12: Plot

6. Conclusion

This project implements the Adaptive Resonance Theory for MNIST datasets. The MNIST datasets were fetched using the Keras Library function. After fetching the MNIST datasets, they were converted from image to binary using grey_to_bin function. In conclusion, the results presented above are the output of implementing the Adaptive Resonance Theory.

7. Reference

- The neural network lecture slides
- https://www.google.com/search?q=adaptive+resonance+theory&rlz=1C1GCEA_enNG992NG993&sxsrf=ALiCzsaTBT1vof6D77kAJMOko0JiSfgotQ:1652126903946&source=lnms&tbm=isch&sa=X&ved=2ahUKEwjm5u6pnNP3AhVmpIsKHVM7DUYQ_AUoAXoECAEQAw&biw=1366&bih=568&dpr=1
- http://www.scholarpedia.org/article/Adaptive_resonance_theory
- https://www.geeksforgeeks.org/adaptive-resonance-theory-art/
- https://www.javatpoint.com/artificial-neural-network-adaptive-resonance-theory