**Auto-Associative Memory**

Deepesh Nathani

dn1005

**Algorithm**

* Set the initial weight matrix to be 0 matrix (all elements 0)
* Now for each input vector to be stored, repeat below steps
  + Set Input Layer Activations to x (Identity Activation Function)
  + Adjust Weights based on Hebb Learning Rule
* Pass the Output layer to below Activation Function

Activation Function (Bipolar Activation Function)

af(x) = {

1, if x > THRESHOLD

0, if –THRESHOLD <= x <= THRESHOLD

-1, if x < -THRESHOLD

}

1. **Analysis of Patterns Stored by Network**

*Cosine Similarity Matrix*

**0 1 2 3 4 5 6 7 8 9**

**0** 1.00 -0.09 0.20 0.66 -0.03 0.37 0.60 0.03 0.71 0.60

**1** -0.09 1.00 0.37 0.03 -0.20 0.09 -0.03 0.09 -0.03 0.09

**2** 0.20 0.37 1.00 0.43 -0.26 0.26 0.14 0.37 0.26 0.37

**3** 0.66 0.03 0.43 1.00 -0.14 0.49 0.60 0.26 **0.83**  0.71

**4** -0.03 -0.20 -0.26 -0.14 1.00 -0.31 -0.20 -0.09 -0.09 -0.09

**5** 0.37 0.09 0.26 0.49 -0.31 1.00 0.77 0.20 0.54 0.66

**6** 0.60 -0.03 0.14 0.60 -0.20 0.77 1.00 0.09 0.77 0.54

**7** 0.03 0.09 0.37 0.26 -0.09 0.20 0.09 1.00 0.09 0.09

**8** 0.71 -0.03 0.26 **0.83** -0.09 0.54 0.77 0.09 1.00 0.77

**9** 0.60 0.09 0.37 0.71 -0.09 0.66 0.54 0.09 0.77 1.00

**Observation**: Pattern-3 and Pattern-8 are very similar, which is also intuitive since both patterns look very similar.

Below is summary of patterns that are stored successfully (with different configurations)

**Columns Description**

*Number of Patterns Trained* => Total patterns passed during training

*Max Number of Patterns Recalled* => Maximum number of Patterns recognized by the network

*Total Combinations with Max Patterns* => Total input combinations that result in maximum recall

(*Threshold* = 0 and *Diagonal elements* not 0)

|  |  |  |
| --- | --- | --- |
| **Number of Patterns Trained** | **Max Number of Patterns Recalled** | **Total Combinations with Max Patterns** |
| 10 | 2 | 1 |
| 9 | 3 | 1 |
| 8 | 4 | 3 |
| 7 | 5 | 3 |
| 6 | 5 | 6 |
| **5** | **5** | **20** |
| 4 | 4 | 71 |
| 3 | 3 | 95 |
| 2 | 2 | 45 |
| 1 | 1 | 10 |

* Maximum patterns recalled = 5
* It matters what combination of patterns we try to store. For instance, see below

|  |  |
| --- | --- |
| Patterns Stored | Patterns Recalled |
| 0, 1, 2, 4, 6, 7, 8, 9 | 1, 4, 7, 8 |
| 0, 1, 3, 4, 5, 7, 8, 9 | 1, 4, 7, 8 |
| 0, 1, 2, 3, 4, 5, 6, 7, 8, 9 | 1, 8 |
| 0, 2, 6 | 0, 2, 6 |

So, it matters, what patterns we try to store since each will have different number of recalled patterns.

* Row 1 and 2 gives patterns that work equally well and produce the same output

*(Threshold = 0 and Diagonal Elements = 0)*

|  |  |  |
| --- | --- | --- |
| **Number of Patterns Trained** | **Max Number of Patterns Recalled** | **Total Combinations with Max Patterns** |
| 10 | 1 | 1 |
| 9 | 2 | 5 |
| 8 | 3 | 1 |
| 7 | 4 | 4 |
| 6 | 5 | 1 |
| **5** | **5** | **8** |
| 4 | 4 | 43 |
| 3 | 3 | 95 |
| 2 | 2 | 45 |
| 1 | 1 | 10 |

*(Bias = -1 or Threshold = 1 and Diagonal Elements != 0)*

|  |  |  |
| --- | --- | --- |
| **Number of Patterns Trained** | **Max Number of Patterns Recalled** | **Total Combinations with Max Patterns** |
| 10 | 2 | 1 |
| 9 | 3 | 1 |
| 8 | 4 | 3 |
| 7 | 4 | 14 |
| 6 | 5 | 6 |
| **5** | **5** | **16** |
| 4 | 4 | 71 |
| 3 | 3 | 85 |
| 2 | 2 | 45 |
| 1 | 1 | 10 |

From the above table, maximum number of patterns that can be successfully stored is 5.

*(Bias = 1 and Threshold = -1 and Diagonal Elements != 0)*

|  |  |  |
| --- | --- | --- |
| **Number of Patterns Trained** | **Max Number of Patterns Recalled** | **Total Combinations with Max Patterns** |
| 10 | 2 | 1 |
| 9 | 3 | 1 |
| 8 | 4 | 3 |
| 7 | 5 | 3 |
| **6** | **6** | **2** |
| 5 | 5 | 20 |
| 4 | 4 | 82 |
| 3 | 3 | 95 |
| 2 | 2 | 45 |
| 1 | 1 | 10 |

From the above table, maximum number of patterns that can be successfully stored is 6.

Now, we know that optimum configuration is to store 6 patterns. In that case, all of them can be recalled successfully. But it does matter what pattern you try to store. For instance, not all combinations (total combinations = 210) can store 6 patterns.

|  |  |
| --- | --- |
| **Patterns You Store** | **Patterns Recalled** |
| 0, 1, 2, 3, 4, 5 | 1, 2, 3, 4 |
| 0, 1, 2, 3, 4, 7 | 0, 1, 3, 4, 7 |
| 0, 1, 2, 3, 5, 6 | 1, 6 |
| 0, 1, 2, 4, 6, 7 | 0, 1, 2, 4, 6, 7 |
| 1, 2, 4, 5, 7, 8 | 1, 2, 4, 5, 7, 8 |

From the above table, it’s evident that not all combinations produce the exact result. Also, there more than one choices can work well (See last two rows).

Orthogonality

* Let’s consider the combination [0, 1, 2, 4, 6, 7]
* From the above similarity matrix, we know that most of them are nearly orthogonal. For instance, look at (0, 1), (0, 2), (1, 6)
* This implies that more orthogonal the patterns, network can recall them easily.

1. **Analysis of Noise Network can handle**

In the earlier section, we saw that the below two combinations of training patterns had maximum recall i.e. maximum number of patterns can be recalled from the network.

* [0, 1, 2, 4, 6, 7]
* [1, 2, 4, 5, 7, 8]

In this section, we will analyze on a network trained from above patterns. Now the idea of noise comes in two forms. We will do experiments on both forms.

* 1. *Error* – When a bit is reversed (1 -> -1 or -1 -> 1)
  2. *Missing Data* – When there is a 0 indicating, unsureness for that bit

Below is the table where an error or missing bit was chosen randomly, and then that pattern was passed through the network. *Pattern Chosen* = [1, 2, 4, 5, 7, 8]

*Columns Description*

*Random Pattern Chosen* – *Random* pattern from the above list chosen

*Total Bits Changed* – Total Number of bits *randomly* modified from the above pattern

*Error* – When Error is induced. ‘True’ if same as original pattern (without error induced)

*Missing* – When missing values is induced. True is same as original pattern

|  |  |  |  |
| --- | --- | --- | --- |
| **Random Pattern Chosen** | **Total Bits Changed** | **Error** | **Missing** |
| 7 | 0 | True | True |
| 4 | 1 | True | True |
| 5 | 2 | True | True |
| 4 | 3 | True | False |
| 1 | 4 | True | True |
| 8 | 5 | True | False |
| 4 | 6 | True | True |
| 8 | 7 | True | True |
| 1 | 8 | False | False |
| 8 | 9 | True | False |
| 2 | 10 | False | True |
| 8 | 11 | False | False |
| 2 | 12 | False | True |
| 2 | 13 | False | False |
| 1 | 14 | False | True |
| 8 | 15 | False | False |
| 5 | 16 | False | False |
| 1 | 17 | False | True |
| 8 | 18 | False | False |
| 8 | 19 | False | True |
| 2 | 20 | False | False |
| 2 | 24 | False | True |
| **7** | **29** | **False** | **True** |
| **7** | **31** | **False** | **True** |

In the above table, we see that the network can handle as much as 31 bits of missing values. For pattern ‘7’ even if 31 bits are turned to ‘0’, even then the network can identify the pattern.

1. **Analysis on Spurious Patterns**

Spurious Patterns are patterns that are recalled from the network, but not trained upon. Below is the description of how spurious patterns were generated for reproducibility. Again, for analysis we use the optimal neural network trained with [1, 2, 4, 5, 7, 8]

* **Negation of all input vectors**. For instance, below is the visualization of ‘1’ and its negation. Negation means flipping all bits in the input vector.

.

. .

. .

. < ---------- Pattern = ‘1’

.

.

. . . . .

. . . .

. . .

. . . < --------- Pattern = Negation of ‘1’

. . . .

. . . .

. . . .

*Columns Description*

*Original Pattern* – Original Pattern

*Does Flipped Pattern come out Same?* – Means when a pattern is flipped then will the network recall the flipped pattern

*Is Spurious Pattern?* – Is the pattern a Spurious Pattern

|  |  |  |
| --- | --- | --- |
| **Original Pattern** | **Does Flipped Pattern come out Same?** | **Is Spurious Pattern?** |
| 1 | Yes | Yes |
| 2 | No | No |
| 4 | Yes | Yes |
| 5 | No | No |
| 7 | Yes | Yes |
| 8 | Yes | Yes |

* Linear Combinations of input vectors. Combinations should be of odd number of input vectors since with even number, there is a possibility of getting 0’s everywhere. Below is the visualization of combination of ‘1’, ‘4’ and ‘7’.

. .

.

.

. < ---------- Looks like pattern ‘2’

.

. . . .

|  |  |  |
| --- | --- | --- |
| **Original Combination** | **Does Flipped Pattern come out Same?** | **Is Spurious Pattern?** |
| 1,2,4 | No | No |
| 1,2,5 | No | No |
| 1,2,7 | No | No |
| 1,2,8 | No | No |
| 4,5,8 | No | No |
| 4,7,8 | No | No |
| 5,7,8 | No | No |

In the above table, you can see that the output from network is not same as the input pattern. But if we visualize the output closely, we see that though it is not the same, it is very similar. For instance, for the above combination (1,4,7) below is the output.

. . .

. . < -------- Similar to what was passed in

.

.

.

.

. . . . .

**CODE**

**import** os  
**import** sys  
**import** pandas  
**import** random  
**import** itertools  
  
**import** numpy **as** np  
**from** scipy **import** spatial  
  
  
INPUT\_FILE\_NAME **=** "TenDigitPatterns.txt"  
INPUT\_PATTERN **=** '#'  
PATTERN\_ROW\_LEN **=** 5  
PATTERN\_VECTOR\_LEN **=** 35  
  
THRESHOLD **= -**1 *# Threshold used at output layer***def readInputFile**(*fileName*)**:** filHandle **=** open(*fileName*)  
 **return** map(**lambda** *pattern***:**pattern[**:**PATTERN\_VECTOR\_LEN], filHandle.readlines())  
  
**def getCosineSimilarity**(*vec1*, *vec2*)**:** *# Length of both vectors is assumed to be same.* **return** round(1 **-** spatial.distance.cosine(*vec1*, *vec2*), 2)  
  
**def constructOrthogonalMatrix**(*patterns*)**:** mat **=** np.zeros(shape **=** (len(*patterns*), len(*patterns*)))  
 counter **=** 0  
 **for** outerPattern **in** *patterns***:** l **=** []  
 **for** innerPattern **in** *patterns***:** l.append(getCosineSimilarity(innerPattern, outerPattern))  
 mat[counter] **=** l  
 counter **=** counter **+** 1  
 **return** mat  
  
**def createVizForPatterns**(*pattern*)**:** lolFunc **= lambda** *lst*, *sz***:** [lst[i**:**i **+** sz] **for** i **in** range(0, len(lst), sz)] *# lambda function* lol **=** lolFunc(*pattern*, PATTERN\_ROW\_LEN)  
  
 **for** row **in** lol**:** rowToDisplay **=** map(**lambda** *ele***:** '.' **if** ele **==** 1 **else** ' ', row)  
 **if**(rowToDisplay **!=** None)**:  
 for** e **in** rowToDisplay**:  
 print** e,  
 **print** "**\n**"  
  
**def induceNoiseInPattern**(*pattern*, *numOfBitsToChange*, *isError*)**:** changedPattern **=** list(*pattern*) *# dont change the original list* randomIndices **=** random.sample(range(0, 34), *numOfBitsToChange*)  
 **for** ran **in** randomIndices**:  
 if**(*isError*)**:** changedPattern[ran] **= -**changedPattern[ran]  
 **else:** changedPattern[ran] **=** 0  
 **return** changedPattern  
  
**def checkIfPatternIsSame**(*patternToTest*, *basePattern*)**:  
 if**(type(*patternToTest*[0]) **!=** type(1.0))**:** patternToTest **=** map(**lambda** *x***:**x.item(), *patternToTest*)  
  
 **if** (len(*patternToTest*) **==** len(*basePattern*))**:** uncommon **=** [1 **for** x, y **in** zip(*patternToTest*, *basePattern*) **if** x **!=** y]  
 **if** (len(uncommon) **==** 0)**: return** True  
 **return** False  
  
**def getNumericNumForPattern**(*pattern*, *patterns*)**:** *# assuming PATTERNS Vector is already filled* match **= -**1  
 **for** patternNum **in** range(0, len(*patterns*))**:  
 if**(checkIfPatternIsSame(*pattern*, *patterns*[patternNum]))**: return** patternNum  
 **return** match  
   
  
**def convertPatternToVector**(*pattern*)**:  
 return** np.array(map(**lambda** *x***:** 1 **if** x **==** INPUT\_PATTERN **else -**1, *pattern*)) *# using bipolar vectors***def convertPatternsToInputVectors**(*patterns*)**:** inputMatrix **=** np.zeros(shape **=** (len(*patterns*), PATTERN\_VECTOR\_LEN))  
 **for** i **in** range(0, len(*patterns*))**:** inputMatrix[i] **=** convertPatternToVector(*patterns*[i]) *# convert '#' pattern to 1 and -1* **return** inputMatrix  
  
'''  
 Activation Function  
 f(x) = {  
 +1 if x > THRESHOLD  
 0 if -THRESHOLD <= x <= THRESHOLD  
 -1 if x < -THRESHOLD  
 }  
'''  
**def runOutputLayerTransferFunc**(*outputFromNet*)**:** out **=** []  
 **for** i **in** range(0, len(*outputFromNet*))**:  
 if**(*outputFromNet*[i] **>** THRESHOLD)**:** out.append(1.0)  
 **elif**(*outputFromNet*[i] **< -**THRESHOLD)**:** out.append(**-**1.0)  
 **else:** out.append(0)  
 **return** out  
  
**def getOutputFromLayer**(*weightMatrix*, *inputPattern*)**:  
 return** *weightMatrix*.dot(*inputPattern*)  
  
**def computeWeightMatrixUsingHebbRule**(*patterns*)**:** weightMatrix **=** np.zeros(shape **=** (PATTERN\_VECTOR\_LEN, PATTERN\_VECTOR\_LEN))  
 **for** pattern **in** *patterns***:** weightMatrix **+=** map(**lambda** *x***:** pattern **if** x **==** 1 **else -**pattern, pattern)  
  
 **return** weightMatrix   
  
**def main**()**:** numbers **=** [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]  
 patterns **=** readInputFile(INPUT\_FILE\_NAME) *# Read the input patterns file* patterns **=** convertPatternsToInputVectors(patterns) *# convert patterns to input-matrix* ''' Cosine Similarity Matrix  
 print pandas.DataFrame(constructOrthogonalMatrix(patterns), numbers, numbers)  
 '''  
  
 '''  
 Part(a)  
 Start with storing all patterns and checking how many can be recalled.  
 Then start removing one pattern and check again. Catch here is that when  
 removing a pattern we need to consider all combinations. So if removing one  
 pattern then (10C9) combinations should be considered.  
  
 results = []  
 for run in range(10, 0, -1):  
 combinations = itertools.combinations(numbers, run)  
 for combination in combinations:  
 inputPatterns = map(lambda x:patterns[x], combination)  
 weightMatrix1 = computeWeightMatrixUsingHebbRule(inputPatterns)

np.fill\_diagonal(weightMatrix1, 0) # removing all self connections  
 output = map(lambda pattern:getNumericNumForPattern(runOutputLayerTransferFunc(  
 getOutputFromLayer(weightMatrix1, pattern)), patterns), patterns)  
 output = set(output)  
 finalOutput = [x for x in output if x != -1]  
 finalOutput.sort()  
 results.append((run, combination, finalOutput))  
  
 for run in [6]:  
 subsetOfResults = [x for x in results if x[0] == run]  
 maxResult = max(subsetOfResults, key=lambda x: len(x[2]))  
 maxResults = [i for i in subsetOfResults if len(i[2]) == len(maxResult[2])]  
  
 print len(maxResults)  
 for res in maxResults:  
 print res  
 '''  
  
 '''  
 Part(b)  
 Analysis of how much noise can the network handle. Optimum Configuration for training = [1, 2, 4, 5, 7, 8]  
 We go systematically here, starting with just one bit error/missing value in patterns. Error can be while  
 training as well as testing. So we analyse both of these scenarios.  
  
 optimalTraining1 = [patterns[1], patterns[2], patterns[4], patterns[5], patterns[7], patterns[8]]  
 weightMatrix1 = computeWeightMatrixUsingHebbRule(optimalTraining1)  
  
 print "Below is for First Optimal Train Patterns"  
 for bitsToChange in range(0, 35, 1):  
 randomPatternToTest = random.choice(optimalTraining1)  
 erroredPattern = induceNoiseInPattern(randomPatternToTest, bitsToChange, True)  
 missingValsPattern = induceNoiseInPattern(randomPatternToTest, bitsToChange, False)  
  
 outputWithErroredPattern = getNumericNumForPattern(runOutputLayerTransferFunc(  
 getOutputFromLayer(weightMatrix1, erroredPattern)), patterns)  
 outputWithMissingValsPattern = getNumericNumForPattern(runOutputLayerTransferFunc(  
 getOutputFromLayer(weightMatrix1, missingValsPattern)), patterns)  
  
 print "Total Bits Changed =",  
 print bitsToChange,  
 print "Random Pattern =",  
 print getNumericNumForPattern(randomPatternToTest, patterns),  
 print " ",  
 print "Output After Error Induced =",  
 print outputWithErroredPattern,  
 print " ",  
 print "Output After Missing Vals Inserted =",  
 print outputWithMissingValsPattern,  
 print "**\n**"  
  
 optimalTraining2 = [patterns[0], patterns[1], patterns[2], patterns[4], patterns[6], patterns[7]]  
 weightMatrix2 = computeWeightMatrixUsingHebbRule(optimalTraining2)  
  
 print "Below is for Second Optimal Train Patterns"  
 for bitsToChange in range(0, 35, 1):  
 randomPatternToTest = random.choice(optimalTraining2)  
 erroredPattern = induceNoiseInPattern(randomPatternToTest, bitsToChange, True)  
 missingValsPattern = induceNoiseInPattern(randomPatternToTest, bitsToChange, False)  
  
 outputWithErroredPattern = getNumericNumForPattern(runOutputLayerTransferFunc(  
 getOutputFromLayer(weightMatrix2, erroredPattern)), patterns)  
 outputWithMissingValsPattern = getNumericNumForPattern(runOutputLayerTransferFunc(  
 getOutputFromLayer(weightMatrix2, missingValsPattern)), patterns)  
  
 print "Total Bits Changed =",  
 print bitsToChange,  
 print "Random Pattern =",  
 print getNumericNumForPattern(randomPatternToTest, patterns),  
 print " ",  
 print "Output After Error Induced =",  
 print outputWithErroredPattern,  
 print " ",  
 print "Output After Missing Vals Inserted =",  
 print outputWithMissingValsPattern,  
 print "**\n**"  
 '''  
  
 '''  
 Part-(c)  
 Analysis on Spurious Patterns. These are patterns that a network recalls, but not in training set.  
 Spurious patterns can be negation of the original pattern or linear combination of input vectors  
 1. Negation of all Input Patterns  
 2. Linear Combinations of Input patterns  
  
  
 optimalTraining = [patterns[1], patterns[2], patterns[4], patterns[5], patterns[7], patterns[8]]  
 weightMatrix = computeWeightMatrixUsingHebbRule(optimalTraining)  
  
 for pattern in optimalTraining:  
 flippedPattern = np.array(-pattern)  
 outputWithSpuriousPattern = runOutputLayerTransferFunc(getOutputFromLayer(weightMatrix, flippedPattern))  
  
 print "Original Pattern =",  
 print getNumericNumForPattern(pattern, patterns)  
 print " ",  
 print "Is the Flipped pattern comes out same as was inserted",  
 print checkIfPatternIsSame(flippedPattern, outputWithSpuriousPattern)  
  
 optimalTraining = [patterns[1], patterns[2], patterns[4], patterns[5], patterns[7], patterns[8]]  
 weightMatrix = computeWeightMatrixUsingHebbRule(optimalTraining)  
  
 allCombinationsOfThreePatterns = itertools.combinations(range(0, 6), 3)  
 for comb in allCombinationsOfThreePatterns:  
 summedPattern = optimalTraining[comb[0]] + optimalTraining[comb[1]] + optimalTraining[comb[2]]  
 #createVizForPatterns(summedPattern)  
  
 outputWithSpuriousPattern = runOutputLayerTransferFunc(getOutputFromLayer(weightMatrix, summedPattern))  
 #createVizForPatterns(outputWithSpuriousPattern)  
 print comb,  
 print " ",  
 print "Is same as Original Pattern =",  
 print checkIfPatternIsSame(summedPattern, outputWithSpuriousPattern)  
 '''  
  
**if** \_\_name\_\_ **==** "\_\_main\_\_"**:** main()