A Hopfield Network for Digits Recognition

Artificial Intelligence Course Project

Gianluca Barbon
818453@stud.unive.it

January 12, 2015

Abstract

This report describes the artificial intelligence course project. This project consists in the implementation of a Hopfield Network using python. The weight matrix will be computed by using different algorithms, in such a way to analyse performances and thus identify the best solution.

1 Introduction

1.1 Neural Networks

They are used for recognition and classification problem, they are capable to learn and so to generalize, that is produce outputs in correspondence of inputs never met before. The training of the net take place presenting a training set (set of example) as input. The answer given by the net for each example will be compared to the desired answer, the difference (or error) between the two will be evaluated and finally the weights will be adjusted by looking at this difference. the process is repeated for the entire training set, until the produced error is minimized, so under a preset threshold.

1.1.1 Classification Problems

Classification problems consists in the classification of an object by looking at its features.

1.2 Hopfield Network [3]

Hopfield networks are neural networks that can be seen as non linear dynamic systems. They are also called recurring networks or feedback networks. We consider neural networks as non linear dynamic systems, where we consider the time variable. In order to do this, we must take into account loops, so we will use recurrent networks. recurrent networks with non linear units are difficult to analyse: they can converge to a stable state, oscillate or follow chaotic trajectories whose behaviour

is not predictable. However, the american physicist J.J. Hopfield discovered that with symmetric connections there will exist a stable global energy function. The Hopfield networks have the following properties:

- **sigle layer recurrent networks** in which each neuron is connected to all the others, with the exception of itself (so no cycles are admitted)

- **symmetric:** the synaptic weight matrix is symmetric, so $W = W^T$. This means that the weights is the same in both direction between two neurons.

- **not linear:** in the continuous formulation, each neuron has a non linear invertible activation function

As for the neuron update, we can use three possible approaches:

- **asynchronous update:** where neurons are updated one by one

- **synchronous update:** all neurons are updated at the same moment

- **continuous update:** all neurons are updated in a continuous way

There exists two formulation of the Hopfield model: the discrete one and the continuous one, that differ for the way in which the time flows. For this project we will use the discrete model. In this model the time flows in discrete way and neurons updates in asynchronous way. as for the neuron input, the McCulloch and Pitts model is used, with the adding of an external influence (or bias?) factor:

$$H_i = \underbrace{\sum_{j \neq i} w_{ij} V_j}_{\text{M\&P model}} + \underbrace{I_i}_{\text{external input}}$$

The activation function is the following:

$$V_i = \begin{cases} +1 & \text{se } H_i > 0 \\ -1 & \text{se } H_i < 0 \end{cases} \quad (1)$$

The update of the neurons is a random process and the selection of the unit to be updated can be done in two ways:

1. at each time instant the unit to be updated is chosen randomly (this mode is useful in simulations)

2. each unit is updated independently with constant probability at each time instant

Unlike feedforward networks, a Hopfield network is a dynamic system. It starts from an initial state

$$\vec{V}(0) = (V_1(0), \ldots, V_n(0))^T$$

and evolves through a trajectory until it reach a fixed point in which $V(t+1) = V(t)$ (convergence). The Hopfield theorem supplies a sufficient condition for the convergence of the system. It uses an energy function E that govern the systems :

$$E = -\frac{1}{2} \sum_{i=1}^{n} \sum_{\substack{j=1 \\ j \neq i}}^{n} w_{ij} V_i V_j - \sum_{i=1}^{n} I_i V_i \qquad (2)$$

Theorem 1.1 (Hopfield theorem: discrete case)**.** *If the weights matrix in a Hopfield network is symmetric, diag(W) = 0, the Energy Function will be a Lyapunov function for the system, so:*

$$\Delta E = E(t+1) - E(t) \leq 0$$

with the equivalence when the system reach a stationary point.

1.3 Learning Algorithms

Learning rule have some characteristics:

- **locality:** a rule can be local, this means that the update of a given weight depends only on informations available to neurons on either side of the connection. Locality provides natural parallelism that is one of the component that makes an Hopfield network a truly parallel machine.

- **incremental:** an incremental rule modifies the old network configuration to memorize a new pattern without needing to refer to any of the previous learnt patterns. This behaviour allows an Hopfield net to be adaptive, thus more suitable for real time situations and changing environments

- **immediate:** an immediate update of the network allows faster learning

- **capacity:** it measure how many patterns can be stored in the network of a given size (number of neurons). Moreover higher capacity allows faster processing times, because the update time is at least proportional to the number of neurons.

1.3.1 Hebb's rule

In contrast with computer's byte-addressable memory, that adopt a precise memory address to locate information, the human brain utilize content-addressable memory, that uses the content of data to locate information. The Hebb rule has been introduced to describe such behaviour and states that the weight between two neurons increases if the two neurons activate simultaneously. In detail, the Hebb postulate says that:

Let us assume that the persistence or repetition of a reverberatory activity (or "trace") tends to induce lasting cellular changes that add to its stability. [...] When an axon of cell A is near enough to excite a cell B and repeatedly or persistently takes part in firing it, some growth process or metabolic change takes place in one or both cells such that A's efficiency, as one of the cells firing B, is increased. [1]

The memory is represented like a set of P patterns x^μ, where $\mu = 1, \ldots, \P$: when a new pattern x is presented, the net typically answers producing the pattern in memory that most resembles to x. Accordingly to the Hebb postulate, proportional weights are used in the activation between a pre and a post synaptic neurons:

$$w_{ij} = \frac{1}{N} \sum_{\mu=1}^{P} x_i^\mu x_j^\mu$$

where N is the number of binary units with output $s_1, \ldots, s_N$. The recall mechanism is the following:

$$s_i = \text{sgn}\left(\sum_j w_{ij} s_j \right)$$

Anyway, there are some problems in the use of Hopfield network as content-addressed memories:

- the maximum number of pattern is $0.15N$

- sometimes the net produces spurious states, that is states that do not belong to the memorized patterns

- the recalled pattern is not necessarily the most similar to the input one

- patterns are not recalled with the same emphasis

1.3.2 Pseudo-Inverse Rule

Pseudo inverse rule try to map a memory to itself before thresholding. In this way the problem can be treated as a real value regression problem, the solution of which is given by the pseudo-inverse. We can consider the Hebb rule as a special case of pseudo-inverse rule for orthogonal training vectors. However, in this rule we have no

local computation and no incremental updates, because it involves the calculation of an inverse.

$$w_{ij} = \frac{1}{N} \sum_{\substack{u=1 \\ v=1}}^{m} x_i^u (Q^{-1})^{uv} x_j^v$$

where m is the total number of patterns and q is:

$$q_{uv} = \frac{1}{N} \sum_{i=1}^{N} x_i^u x_j^v$$

This rule, also called projection learning rule [2], allows to improve the retrieval capability of the net with respect to Hebb rule, bringing the maximum number of pattern to $0.5N$.

1.3.3 Storkey Rule [4]

Even if the pseudo-inverse rule performs better than the Hebbian one, it is never incremental and local only if not immediate. Storkey propose an algorithm to increase the capacity of the Hebbian rule without losing locality and incrementality.

$$w_{ij}^0 = 0 \; \forall \; i,j \; \text{ and }$$

$$w_{ij}^\nu = w_{ij}^{\nu-1} + \frac{1}{N} x_i^\nu x_j^\nu - \frac{1}{N} x_i^\nu h_{ji}^\nu - \frac{1}{N} h_{ij}^\nu x_j^\nu$$

where h_{ij} is a form of local field at neuron i:

$$h_{ij}^\mu = \sum_{\substack{k=1 \\ k \neq i,j}}^{n} w_{ij}^{\mu-1} x_k^\mu$$

2 Implementation

2.1 Learning Algorithms

In the pseudo inverse implementation the Q matrix is already returned as an inverse, in order to save computational time. The original algorithm that iterated the units for the sum computation has been substituted with a dot product operator thanks to numpy library.

Storkey learning rule. The sub function that compute c has not been optimized, because the result with the dot product where too bad with respect to the normal one. As for the core Storkey formula, it has been implemented using loops. Improvement where not possible, because dot product could not be exploited.

3 Results

4 Conclusions

References

[1] D. O. Hebb. *The Organization of Behavior*. Wiley, New York, 1949.

[2] I. G. L. Personnaz and G. Dreyfus. Collective computational properties of neural networks: New learning mechanisms. *Phys. Rev. A*, 1986.

[3] M. Pelillo. Artificial intelligence course, 2014.

[4] A. Storkey. Increasing the capacity of a hopfield network without sacrificing functionality. In *ICANN97: Lecture Notes in Computer Science 1327*, pages 451–456. Springer-Verlag, 1997.

A Appendix: Code

A.1 HopfieldNet.py

```python
__author__ = 'gbarbon'

import numpy as np
import random as rd
import trainers as tr


class HopfieldNet:

    def __init__(self, train_input, trainer_type, image_dimensions):

        #number of training patterns and number of units
        n_patterns = train_input.shape[0]
        self.n_units = train_input.shape[1]

        # crating threshold array (each unit has its own threshold)
        self.threshold = np.zeros(self.n_units)

        # setting image dimension
        self.image_dimensions = image_dimensions

        # net training
        if trainer_type == "hebbian":
            self.weights = tr.hebb_train(train_input, n_patterns, self.n_units)
        elif trainer_type == "pseudoinv":
            self.weights = tr.pseudo_inverse_train(train_input, n_patterns, self.n_units)
        #else:

    def single_unit_updater(self, unit_idx, pattern):

        #temp = sum(weights[unit_idx,:]*pattern[:]) - threshold[unit_idx]
        # we implement a powerful version that uses dotproduct with numpy
        temp = np.dot(self.weights[unit_idx, :], pattern[:]) - self.threshold[unit_idx]
        if temp >= 0:
            # pattern[unit_idx] = 1
            return 1
        else:
            # pattern[unit_idx] = 0
            return -1

    def energy(self, pattern):
        e = 0
        length = len(pattern)
        for i in range(length):
            for j in range(length):
                if i == j:
                    continue
                else:
                    e += self.weights[i][j] * pattern[i] * pattern[j]

        sub_term = np.dot(self.threshold, pattern)
        e = -1 / 2 * e - sub_term
        return e

    # function overloading in threshold
```

```python
56        def test(self, pattern, threshold=0):
57
58            #setting threshold if threshold != 0
59            if threshold != 0:
60                self.threshold = threshold
61
62            pattern = pattern.flatten()  # flattening pattern
63            energy = self.energy(pattern)  # energy init
64
65            k = 0
66            while k < 10:
67                randomrange = range(len(pattern))
68                rd.shuffle(randomrange)
69                for i in randomrange:
70                    pattern[i] = self.single_unit_updater(i, pattern)
71                    #nota: l'energia deve essere calcolata ad ogni cambiamento di una singola unita' o d
72                temp_e = self.energy(pattern)
73                if temp_e == energy:
74                    #break while loop
75                    pattern.shape = (self.image_dimensions[0], self.image_dimensions[1])
76                    return pattern
77                else:
78                    energy = temp_e
79                k += 1
80
81            pattern.shape = (self.image_dimensions[0], self.image_dimensions[1])
82            return pattern
```

A.2 trainers.py

```python
1   __author__ = 'gbarbon'
2
3   import numpy as np
4
5
6   # train intput is in the form of a vector
7   def hebb_train(train_input, n_patterns, n_units):
8       # weights matrix init to zeros
9       weights = np.zeros((n_units, n_units))
10
11      # l
12      for l in range(n_patterns):
13          for i in range(n_units):
14              for j in range(n_units):
15                  if i == j:
16                      continue
17              # weights[i,j] = (1/n_units)* sum(...)
18                  else:
19                      weights[i, j] += train_input[l, i] * train_input[l, j]
20                  # print("Temp weight at this point is", weights[i,j])
21                  # uguale???: weights[i,j] = sum(train_input[:,i]*train_input[:,j])
22
23      return weights
24
25  #  uses the storkey rule
26  def storkey_train(train_input, n_patterns, n_units):
27      weights = np.zeros((n_units, n_units))
28
29      #..#
30
31      return weights
32
33  def q_pseudo_inv(train_input, n_patterns, n_units):
34      q = np.zeros((n_patterns, n_patterns))
35
36      for v in range(n_patterns):
37          for u in range(n_patterns):
38              # for i in range(n_units):
39              #     q[u][v] += train_input[v][i]*train_input[u][i]
40              q[u][v] = np.dot(train_input[v], train_input[u])
41
42      q *= 1 / float(n_units)
43      q = np.linalg.inv(q) # inverseof the matrix
44
45      return q
46
47  # uses the pseudo inverse training rule
48  def pseudo_inverse_train(train_input, n_patterns, n_units):
49      weights = np.zeros((n_units, n_units))
50
51      q = q_pseudo_inv(train_input, n_patterns, n_units)
52
53      for v in range(n_patterns):
54          for u in range(n_patterns):
55              for i in range(n_units):
56                  for j in range(n_units):
57                      if i == j:
```

```
58                              continue
59                       else:
60                           weights[i, j] += train_input[v, i] * q[v][u] *train_input[u, j]
61
62
63      weights *= 1 / float(n_units)
64      return weights
```

A.3 imageManager.py

```python
1   __author__ = 'gbarbon'
2
3   from PIL import Image
4   from PIL import ImageFilter
5   # from PIL import ImageEnhance
6   import os
7   import numpy as np
8
9
10  # NOTE: in images dimension are: first:= columns, second:= rows
11
12  def image_cropper(image, new_dimensions):
13      width, height = image.size  # Get dimensions
14
15      left = (width - new_dimensions[1]) / 2
16      top = (height - new_dimensions[0]) / 2
17      right = (width + new_dimensions[1]) / 2
18      bottom = (height + new_dimensions[0]) / 2
19
20      return image.crop((left, top, right, bottom))
21
22
23  def image_resizer(image, new_dimensions):
24      return image.resize(new_dimensions)
25
26
27  def to_greyscale(image):
28      return image.convert("L")
29
30
31  def to_blackwhite(input_image):
32      return input_image.convert("1")
33
34
35  # convert image to a matrix of 0 and 1 and to black and white
36  def tomatrix_bew(image):
37      imarray = np.array(image.getdata(), np.uint8).reshape(image.size[1], image.size[0])
38      row = imarray.shape[0]
39      cols = imarray.shape[1]
40      newarray = np.zeros((row, cols))
41
42      for i in range(row):
43          for j in range(cols):
44              if imarray[i][j] <= 127:
45                  newarray[i][j] = 1
46              else:
47                  newarray[i][j] = 0
48      return newarray
49
50
51  # convert image to a matrix of 0 and 1
52  def tomatrix(image):
53      matrix = np.array(image.getdata(), np.uint8).reshape(image.size[1], image.size[0])
54      rows = matrix.shape[0]
55      cols = matrix.shape[1]
56      newarray = np.zeros((rows, cols))
57
```

```python
58          for i in range(rows):
59              for j in range(cols):
60                  if matrix[i][j] == 0:
61                      newarray[i][j] = 1
62
63          return newarray
64
65
66  def collectimages(finaldim, img_dir):
67      i = 0
68      entries = 0
69
70      # number of files checking
71      for img_file in os.listdir(img_dir):
72          if img_file.endswith(".tiff"):
73              entries += 1
74
75      dataset = np.zeros((entries, finaldim[0] * finaldim[1]))
76
77      for img_file in os.listdir(img_dir):
78          if img_file.endswith(".tiff"):
79              newdir = img_dir + "/" + img_file
80              im = Image.open(newdir)
81
82              orig_dim = im.size
83
84              # Image conversion to black and white
85              imp = to_greyscale(im)
86              imp = to_blackwhite(imp)
87
88              # Image filtering
89              imp = imp.filter(ImageFilter.MedianFilter(size=5))
90              #imp = imp.filter(ImageFilter.ModeFilter(size=5))
91
92              if orig_dim[0] > 100 and orig_dim[1] > 100:
93                  # crop if dimensions higher than 100
94                  imp = image_cropper(imp, [100, 100])
95                  imp = image_resizer(imp, finaldim)
96              else:
97                  imp = image_resizer(imp, [40, 70])
98                  imp = image_resizer(imp, finaldim)
99
100                 #imp.show()  # shows image in external program
101                 imarray = tomatrix(imp)
102                 dataset[i] = imarray.flatten()
103                 i += 1
104
105     return dataset
```

A.4 utils.py

```
1   __author__ = 'gbarbon'
2
3   from matplotlib import pyplot as plt
4   import random as rnd
5   import copy as cp
6
7
8   # convert images from 0/1 to -1/1
9   def image_converter(input_image):
10      image = cp.copy(input_image)
11      image *= 2
12      image -= 1
13      return image
14
15
16  # corrupts images
17  def corrupter(input_image, corrupt_param):
18      dim_row = input_image.shape[0]
19      dim_col = input_image.shape[1]
20      corrupted_image = cp.copy(input_image).flatten()
21
22      for i in range(corrupt_param):
23          corr_idx = rnd.randint(0, (dim_row*dim_col-1))
24          corrupted_image[corr_idx] *= -1
25      corrupted_image.shape = (dim_row, dim_col)
26
27      return corrupted_image
28
29
30  # Plot the results
31  def plotter(test_set, result_set):
32      ntest = len(test_set)
33      k = 1
34      for i in range(ntest):
35          plt.subplot(ntest, 2, k)
36          plt.imshow(test_set[i], interpolation="nearest")
37          k += 1
38          plt.subplot(ntest, 2, k)
39          plt.imshow(result_set[i], interpolation="nearest")
40          k += 1
41      plt.show()
```

A.5 tests.py

```
1  __author__ = 'gbarbon'
2
3  import numpy as np
4  import HopfieldNet
5  import utils as utl
6  import imageManager as iM
7
8
9  def test1():
10     # Create the training patterns
11     a_pattern = np.array([[0, 0, 0, 1, 0, 0, 0],
12                           [0, 0, 1, 0, 1, 0, 0],
13                           [0, 1, 0, 0, 0, 1, 0],
14                           [0, 1, 1, 1, 1, 1, 0],
15                           [0, 1, 0, 0, 0, 1, 0],
16                           [0, 1, 0, 0, 0, 1, 0],
17                           [0, 1, 0, 0, 0, 1, 0]])
18
19     b_pattern = np.array([[0, 1, 1, 1, 1, 0, 0],
20                           [0, 1, 0, 0, 0, 1, 0],
21                           [0, 1, 0, 0, 0, 1, 0],
22                           [0, 1, 1, 1, 1, 0, 0],
23                           [0, 1, 0, 0, 0, 1, 0],
24                           [0, 1, 0, 0, 0, 1, 0],
25                           [0, 1, 1, 1, 1, 0, 0]])
26
27     c_pattern = np.array([[0, 1, 1, 1, 1, 1, 0],
28                           [0, 1, 0, 0, 0, 0, 0],
29                           [0, 1, 0, 0, 0, 0, 0],
30                           [0, 1, 0, 0, 0, 0, 0],
31                           [0, 1, 0, 0, 0, 0, 0],
32                           [0, 1, 0, 0, 0, 0, 0],
33                           [0, 1, 1, 1, 1, 1, 0]])
34
35     a_pattern = utl.image_converter(a_pattern)
36     b_pattern = utl.image_converter(b_pattern)
37     c_pattern = utl.image_converter(c_pattern)
38
39     train_input = np.array([a_pattern.flatten(), b_pattern.flatten(), c_pattern.flatten()])
40
41     #hebbian training
42     net = HopfieldNet.HopfieldNet(train_input, "hebbian", [7, 7])
43
44     # creating test set
45     a_test = utl.corrupter(a_pattern, 5)
46     b_test = utl.corrupter(b_pattern, 5)
47     c_test = utl.corrupter(c_pattern, 5)
48
49     # training and testing the net
50     a_result = net.test(a_test)
51     b_result = net.test(b_test)
52     c_result = net.test(c_test)
53
54     #Show the results
55     test_set = np.array([a_test, b_test, c_test])
56     result_set = np.array([a_result, b_result, c_result])
57     utl.plotter(test_set, result_set)
```

```python
58
59
60    def test2 ():
61        images_dir = "/Users/jian/Dropbox/AI_dropbox/progetto_2014/dummy_data_set/courier_digits_data_set
62        dim = [14, 9]  # in the form rows * cols
63        testel = 8  # elements for training
64        corruption_val = 30
65
66        image_dim = [dim[1], dim[0]]  # changing shape for images
67
68        # Loading images data set
69        temp_train = iM.collectimages(image_dim, images_dir)
70
71        train_input = np.zeros((testel, dim[0] * dim[1]))
72        for i in range(testel):
73            train_input[i] = temp_train[i]
74
75        # image conversion to 1 and -1 for Hopfield net
76        for i in range(train_input.shape[0]):
77            temp = utl.image_converter(train_input[i].reshape(dim))
78            train_input[i] = temp.flatten()
79
80        # training the net
81        net = HopfieldNet.HopfieldNet(train_input, "hebbian", dim)
82
83        # testing the net
84        test_set = np.zeros((testel, dim[0], dim[1]))
85        result_set = np.zeros((testel, dim[0], dim[1]))
86        for i in range(testel):
87            test_set[i] = utl.corrupter(train_input[i].reshape(dim), corruption_val)
88            result_set[i] = net.test(test_set[i])
89
90        # Plotting results
91        utl.plotter(test_set, result_set)
92
93    def test3 ():
94        images_dir = "/Users/jian/Dropbox/AI_dropbox/progetto_2014/dummy_data_set/digital7_digit_data_set
95        dim = [25, 16]  # in the form rows * cols
96        testel = 10  # elements for training
97        corruption_val = 10
98
99        image_dim = [dim[1], dim[0]]  # changing shape for images
100
101       # Loading images data set
102       temp_train = iM.collectimages(image_dim, images_dir)
103
104       train_input = np.zeros((testel, dim[0] * dim[1]))
105       for i in range(testel):
106           train_input[i] = temp_train[i]
107
108       # image conversion to 1 and -1 for Hopfield net
109       for i in range(train_input.shape[0]):
110           temp = utl.image_converter(train_input[i].reshape(dim))
111           train_input[i] = temp.flatten()
112
113       # training the net
114       net = HopfieldNet.HopfieldNet(train_input, "pseudoinv", dim)
115
116       # testing the net
```

```python
117        test_set = np.zeros((testel, dim[0], dim[1]))
118        result_set = np.zeros((testel, dim[0], dim[1]))
119        for i in range(testel):
120            test_set[i] = utl.corrupter(train_input[i].reshape(dim), corruption_val)
121            result_set[i] = net.test(test_set[i])
122
123        # Plotting results
124        utl.plotter(test_set, result_set)
125
126 test3()
```