A Hopfield Network for Digits Recognition

Artificial Intelligence Course Project

Gianluca Barbon
818453@stud.unive.it

January 15, 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 discrite 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:

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

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 :

$$E = -\frac{1}{2}\sum_{i=1}^{n}\sum_{\substack{j=1 \\ j\neq i}}^{n} w_{ij}V_iV_j - \sum_{i=1}^{n} I_iV_i \tag{2}$$

Theorem 1.1 (Hopfield theorem: discrete case)**.** *If the weights matrix in a Hopfield network is symmetric, $\mathrm{diag}(W) = 0$, 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 $\mu = 1, \ldots, ¶$: 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:

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

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 [5]

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:

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

1.4 Network Capacity

2 Implementation

The Hopfield network is implemented with the HopfieldNet class. Thus, a network will be an instance of this class. When an HopfieldNet object is created, the net is immediately initialized with the given input and the given training algorithm. The class is provided with a test function that performs the test of the whole net with the given test set. This function uses a neuron activation function, called *single_unit_updater*, that operates on a single unit. The behaviour of the net is governed by the energy function, called by the test function at each iteration. Units to be updated are chosen randomly; when every units has been updated, the energy is computed and compared to the one obtained at time $t-1$. When the difference between the two is zero, the update phase is stopped and result is returned to the user.

2.1 Learning Algorithms

2.1.1 Hebbian rule

The Hebbian rule has been implemented in three different ways. The first one is the original algorithm, with three nested loops: the outer one to iterate over training patterns, while the internal one iterate over the elements of the matrix. This solution, while perfectly follows the algorithm, does not performs wells, so it has been decided to exploits the strengths of the python language to improve performances. Following this idea, solution two uses the *dot product* function provided by python in order to compute the product between points of a pattern and the sum of the corresponding results for each point between different patterns, thus avoiding the outer loop. This advantage is better exploited in solution three (the one adopted in the definitive version), where also the two loops that iterates over elements are removed. This solution perform directly the dot products between the transposed input patterns matrix and the normal one. Anyway, even if solution three is the best one, it has been discovered that solution one and two are the best choice for very small images.

2.1.2 Pseudo-inverse rule

The pseudo-inverse rule has not been improved with predefined python functions, because of the complexity of the formula. Thus, the pseudo inverse training algorithm is implemented with four nested loops: the external ones iterate over training patterns while the internal ones iterate over single units for the weight computation. Moreover, this algorithm exploits a separate function for the computation of the Q matrix. The original algorithm that iterated the units for the sum computation has been substituted with a dot product operator thanks to numpy library. Finally, in the pseudo inverse implementation the Q matrix is already returned as an inverse, in order to save computational time.

2.1.3 Storkey's rule

The core Storkey formula has been implemented using loops. Improvement where not possible, because dot product could not be exploited. As for the pseudo-inverse alg. also this one uses an external function, the *h_storkey* function, that computes h. Also this function has not been optimized, even if this was possible, because the result with the dot product where too bad with respect to the normal one.

Notice that both pseudo inverse and Storkey avoid the computation of half of the matrix, by taking advantage of the fact that the matrix is symmetric.

2.2 Minor functions

A lot of minor functions have been implemented in order to support the program. Dataset loading functions have been provided, as well as plotter function to show results. Two interesting functions, that resulted very useful in the test phase, are the following ones:

2.2.1 Corruption and Erase functions

In order to test the capacity of the net to recognize corrupted images, two function have been implemented:

- **corrupter** this function corrupt the images by introducing noise, represented by 1 values. A parameter allows to decide the percentage of image to be noised. Involved points are randomly distributed in the image.
- **image_eraser** this function will erase (fill with "1" values) a part of the image. The percentage of the deletion is decided through a parameter.

3 Testing the net

3.1 Data set

These section describes the dataset used in the project. Each dataset has been used for both training and testing phases, with the exception of the Semeion, which has been used only for testing.

- **Courier Font Digits** This dataset has been created for this project. It is composed of ten tiff images of $44x70$ pixels in RGB colormap, with white background. Each image contain a different digit written in black Courier font, from 0 to 9. These images come from $100x100$ pixels images, where lateral white spaces have been removed in order to improve algorithm efficiency.
- **Digital 7 Font Digits** Also this dataset has been created for this project. Like the Courier data set, it is composed of ten tiff images, where each one contains a different digit written in black Digital 7 font, from 0 to 9. The image dimension is of $44x70$ pixels. As the Courier data set images are in RGB colormap, with white background.
- **Semeion Handwritten Data Set** This dataset was created by Tactile Srl and donated in 1994 to Semeion Research Center of Sciences of Communication for machine learning research. The data set consists of 1593 scanned handwritten digits from around 80 persons, stretched in a rectangular box 16x16 in a gray scale of 256 values. Then each pixel of each image was scaled into a bolean (1/0) value using a fixed threshold. Each person wrote on a paper all the digits from 0 to 9, twice (the first time in the normal way while the second one in a fast way)[4].

3.2 Results

The network has been tested with the three different learning algorithms: Hebbian, Pseudo-inverse and Storkey. The most relevant tests are the ones performed with the use of the Courier Font Digits dataset, with an image dimension of $14x9$ that leads to a network of 126 units. The Pseudo-Inverse is resulted the best learning algorithm, with an accuracy of 100%. The Storkey resulted second with an accuracy of $98,7\%$ while the last one is resulted the Hebbian rule, performing only an accuracy of $74,4\%$. On the contrary, as for time performances, the Hebbian resulted the best one, by training a network with 10 patterns in just 0.0002 seconds. Pseudo-inverse rule obtained 1.5 seconds for the same number of patterns, while the Storkey's resulted the worst algorithm, computing the training in about 21 seconds. It is important to notice that Hebbian implementation exploits the dot product function supplied by python, while the Storkey's rule could not be improved because of its complexity.

Some tests have been developed to check the ability of the network to recall stored memories with the use of corrupted or partial images. Results in this case are very interesting. In example with the corruption of the 20% of each test images and the deletion of the 30% of the same images, the Pseudo-inverse algorithm still obtained an high accuracy, correctly matching the 89% of the images. Also in this case, the Pseudo-inverse resulted the best learning algorithm.

Further tests have been done with the use of the Digital 7 Font Digits data set. In this case the chosen dimension is of $25x16$, leading to a network of 400 units. The aim of this test was to check the network behaviour with larger and clearer images. Anyway, even if images in this data set appear to be clearer to a human, the network results instead increased the number of errors. This is due to the fact that this font uses digits that are very similar. In example, vertical lines are used in almost all the numbers, and this lead to an incorrect match by the Hopfield net.

Finally a set of tests has been performed also with the use of the Courier Font Digits dataset as training and the Semeion Handwritten Data Set as test set. The dimension of the training patterns has been converted to

16x16 in order to match the Semeion one, thus bringing the total number of units to 256. Even if such kind of test could be quite strange (the numbers used in the test are handwritten while the ones used for training come from a computer font), results are interesting: indeed, the network appears to be able to recognize some of the handwritten digits. This is also due to the fact that the Courier font used in training images fits well handwritten digits. The fact that the Courier Font Digits set images have been stretched to match the Semeion images also contributed to positive results. Finally, it is important to notice that handwritten digits are picked randomly from the dataset.

3.2.1 Images filtering

Hebbian rule results are improved by adding a filter in the image sampling before the network training. This filter is a median filter provided by the Python Image Library, and has been inserted after the conversion of the image to black and white. This kind of filter is generally used to perform noise reduction in images, and appeared to be very useful in the image file reading/conversion by filling white pixels or removing useless pixels near the images edges. Results confirmed these behaviours: with the hebbian training algorithm, there were an improvement of about 9% in correct result (with respect to results without the use of the filtering).

4 Results and Conclusions

The Pseudo-Inverse rule algorithm resulted to be the best learning rule, even if it does not respect Importance of the training images: images that are very different between them performs better.
Increase of the number of units (and thus image dimension) does not affect results.

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 notes, 2014.

[4] T. s.r.l. Semeion handwritten data set, 1994.

[5] 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

```
__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)
            self.weights = tr.hebb_train(train_input, self.n_units)
        elif trainer_type == "pseudoinv":
            self.weights = tr.pseudo_inverse_train(train_input, n_patterns, self.n_units)
        elif trainer_type == "storkey":
            self.weights = tr.storkey_train(train_input, n_patterns, self.n_units)
        elif trainer_type == "sanger":
            self.weights = tr.sanger_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]

```

```
56          sub_term = np.dot(self.threshold, pattern)
57          e = -1 / 2 * e - sub_term
58          return e
59
60      # function overloading in threshold
61      def test(self, pattern, threshold=0):
62
63          #setting threshold if threshold != 0
64          if threshold != 0:
65              self.threshold = threshold
66
67          pattern = pattern.flatten()  # flattening pattern
68          energy = self.energy(pattern)  # energy init
69
70          k = 0
71          while k < 10:
72              randomrange = list(range(len(pattern)))
73              rd.shuffle(randomrange)
74              for i in randomrange:
75                  pattern[i] = self.single_unit_updater(i, pattern)
76                  #nota: l'energia deve essere calcolata ad ogni cambiamento di una singola unita' o d
77              temp_e = self.energy(pattern)
78              if temp_e == energy:
79                  #break while loop
80                  pattern.shape = (self.image_dimensions[0], self.image_dimensions[1])
81                  return pattern
82              else:
83                  energy = temp_e
84              k += 1
85
86          pattern.shape = (self.image_dimensions[0], self.image_dimensions[1])
87          return pattern
```

A.2 trainers.py

```
__author__ = 'gbarbon'

import numpy as np
import time


# train intput is in the form of a vector
# def hebb_train(train_input, n_patterns, n_units):
def hebb_train(train_input, n_units):
    # weights matrix init to zeros

    # # 1: original hebb algorithm
    # start = time.time()
    # w1 = np.zeros((n_units, n_units))
    # start = time.time()
    # for l in range(n_patterns):
    # for i in range(n_units):
    # for j in range(i+1, n_units): # in order to compute only the upper half matrix
    # w1[i, j] += train_input[l, i] * train_input[l, j]
    #             w1[j, i] = w1[i, j]
    # w1 *= 1 / float(n_units)
    # end = time.time()
    # elapsed = end - start
    # print("w1 elapsed time", elapsed)

    # # 2: hebb rule improved substituting the pattern loop with dot products
    # start = time.time()
    # w2 = np.zeros((n_units, n_units))
    # train_transp = zip(*train_input)
    # for i in range(n_units):
    #     for j in range(i+1, n_units): # in order to compute only the upper half matrix
    #         w2[i, j] = np.dot(train_transp[i], train_transp[j])
    #         w2[j, i] = w2[i, j]
    # w2 *= 1 / float(n_units)
    # end = time.time()
    # elapsed = end - start
    # print("w2 elapsed time", elapsed)

    # 3: hebb rule improved with matrix multiplication
    start = time.time()
    train_transp = zip(*train_input)  # matrix transpose
    train_transp = np.transpose(train_input)  # matrix transpose
    #train_transp = train_transp.astype(np.float64)
    #w3 = np.zeros((n_units, n_units))
    w3 = np.dot(train_transp, train_input)
    w3 = w3.astype(float)
    w3 *= 1 / float(n_units)
    np.fill_diagonal(w3, 0)
    end = time.time()
    elapsed = end - start
    #print("w3 elapsed time", elapsed)
    print("Hebbian elapsed time", elapsed)

    # # Checking issues
    # if np.array_equal(w1,w2) and np.array_equal(w2, w3):
    #     print("All methods returns the same weight matrix.")

```

```
58      # else:
59      #     print("w1 and w2 ", np.array_equal(w1,w2))
60      #     print("w2 and w3 ", np.array_equal(w2,w3))
61      #     print("w3 and w1 ", np.array_equal(w1,w3))
62
63      return w3
64
65
66  # support function fro pseudo inverse rule
67  def q_pseudo_inv(train_input, n_patterns, n_units):
68      # # Original version with loop
69      # start = time.time()
70      # q1 = np.zeros((n_patterns, n_patterns))
71      # for v in range(n_patterns):
72      # for u in range(n_patterns):
73      # for i in range(n_units):
74      #             q1[u][v] += train_input[v][i] * train_input[u][i]
75      # end = time.time()
76      # elapsed = end - start
77      # print("original version elapsed time", elapsed)
78
79      # New version with dot product
80      # start = time.time()
81      q2 = np.zeros((n_patterns, n_patterns))
82      for v in range(n_patterns):
83          for u in range(n_patterns):
84              q2[u][v] = np.dot(train_input[v], train_input[u])
85      # end = time.time()
86      # elapsed = end - start
87      # print("improved version elapsed time", elapsed)
88
89      # # Checking issues
90      # if np.array_equal(q1, q2):
91      #     print("Both methods returns the same Q matrix.")
92
93      q2 *= 1 / float(n_units)
94      q = np.linalg.inv(q2)  # inverseof the matrix
95
96      return q
97
98
99  # uses the pseudo inverse training rule
100 def pseudo_inverse_train(train_input, n_patterns, n_units):
101     weights = np.zeros((n_units, n_units))
102
103     start = time.time()
104     # notice: the matrix is returned already inverse
105     q = q_pseudo_inv(train_input, n_patterns, n_units)
106
107     for v in range(n_patterns):
108         for u in range(n_patterns):
109             for i in range(n_units):
110                 for j in range(i + 1, n_units):  # in order to compute only the upper half matrix
111                     weights[i, j] += train_input[v, i] * q[v][u] * train_input[u, j]
112                     weights[j, i] = weights[i, j]
113     weights *= 1 / float(n_units)
114
115     end = time.time()
116     elapsed = end - start
```

```
117     print("Pseudo inverse elapsed time", elapsed)
118
119     return weights
120
121
122 # support function for storkey rule
123 def h_storkey(weights, i_index, j_index, pattern, pattern_idx, n_units):
124     h = 0
125
126     # start = time.time()
127     for k in range(n_units):
128         if k != i_index and k != j_index:
129             h += weights[i_index][k] * pattern[pattern_idx][k]
130     # end = time.time()
131     # elapsed1 = end - start
132
133     # start = time.time()
134     # h2 = 0
135     # h2 = np.dot(weights[i_index], pattern[pattern_idx])
136     # end = time.time()
137     # h2 -= weights[i_index][i_index]*pattern[pattern_idx][i_index]
138     # h2 -= weights[i_index][j_index]*pattern[pattern_idx][j_index]
139     # # end = time.time()
140     # elapsed2 = end - start
141     #
142     # if h==h2:
143     # print("Results are the same")
144     # else:
145     # print("RESULTS ARE DIFFERENT!! h1: ", h ," h2: ", h2)
146     # print("First perf is ", elapsed1," while dot is ", elapsed2)
147
148     return h
149
150
151 # uses the storkey rule
152 def storkey_train(train_input, n_patterns, n_units):
153     weights = np.zeros((n_units, n_units))
154
155     start = time.time()
156     for l in range(n_patterns):
157         for i in range(n_units):
158             for j in range(i + 1, n_units):  # in order to compute only the upper half matrix
159                 temp = train_input[l, i] * train_input[l, j]
160                 temp -= train_input[l, i] * h_storkey(weights, j, i, train_input, l, n_units)
161                 temp -= h_storkey(weights, i, j, train_input, l, n_units) * train_input[l, j]
162                 temp *= 1 / float(n_units)
163                 weights[i, j] += temp
164                 weights[j, i] = weights[i, j]
165         print("Storkey rule iteration: ", l)
166     end = time.time()
167     elapsed = end - start
168     print("Storkey algorithm execution time: ", elapsed)
169
170     return weights
171
172
173 # sanger rule applied to hopfield networks, probably not working
174 def sanger_train(train_input, n_patterns, n_units):
175     weights = np.zeros((n_units, n_units))
```

```
176
177         start = time.time()
178         for l in range(n_patterns):
179             for i in range(n_units):
180                 for j in range(i + 1, n_units):
181                     temp = 0
182                     for k in range(j):
183                         temp += weights[i][k] * train_input[l][k]
184                     weights[i][j] = train_input[l][j] * train_input[l][i] - train_input[l][j] * temp
185                     weights[i][j] *= 1 / float(n_units)
186                     weights[j][i] = weights[i][j]
187
188         end = time.time()
189         elapsed = end - start
190         print("Sanger algorithm execution time: ", elapsed)
191
192         return weights
193     \subsection{imageManager.py}
194     __author__ = 'gbarbon'
195
196     from PIL import Image
197     from PIL import ImageFilter
198     # from PIL import ImageEnhance
199     import os
200     import numpy as np
201
202
203     # NOTE: in images dimension are: first:= columns, second:= rows
204
205     def image_cropper(image, new_dimensions):
206         width, height = image.size  # Get dimensions
207
208         left = (width - new_dimensions[1]) / 2
209         top = (height - new_dimensions[0]) / 2
210         right = (width + new_dimensions[1]) / 2
211         bottom = (height + new_dimensions[0]) / 2
212
213         return image.crop((left, top, right, bottom))
214
215
216     def image_resizer(image, new_dimensions):
217         return image.resize(new_dimensions)
218
219
220     def to_greyscale(image):
221         return image.convert("L")
222
223
224     def to_blackwhite(input_image):
225         return input_image.convert("1")
226
227
228     # convert image to a matrix of 0 and 1 and to black and white
229     def tomatrix_bew(image):
230         imarray = np.array(image.getdata(), np.uint8).reshape(image.size[1], image.size[0])
231         row = imarray.shape[0]
232         cols = imarray.shape[1]
233         newarray = np.zeros((row, cols))
234
```

```
235     for i in range(row):
236         for j in range(cols):
237             if imarray[i][j] <= 127:
238                 newarray[i][j] = 1
239             else:
240                 newarray[i][j] = 0
241     return newarray
242 
243 
244 # convert image to a matrix of 0 and 1
245 def tomatrix(image):
246     matrix = np.array(image.getdata(), np.uint8).reshape(image.size[1], image.size[0])
247     rows = matrix.shape[0]
248     cols = matrix.shape[1]
249     newarray = np.zeros((rows, cols))
250 
251     for i in range(rows):
252         for j in range(cols):
253             if matrix[i][j] == 0:
254                 newarray[i][j] = 1
255 
256     return newarray
257 
258 
259 def collectimages(finaldim, img_dir, filter):
260     i = 0
261     entries = 0
262 
263     # number of files checking
264     for img_file in os.listdir(img_dir):
265         if img_file.endswith(".tiff"):
266             entries += 1
267 
268     dataset = np.zeros((entries, finaldim[0] * finaldim[1]))
269 
270     for img_file in os.listdir(img_dir):
271         if img_file.endswith(".tiff"):
272             newdir = img_dir + "/" + img_file
273             im = Image.open(newdir)
274 
275             orig_dim = im.size
276 
277             # Image conversion to black and white
278             imp = to_greyscale(im)
279             imp = to_blackwhite(imp)
280 
281             # Image filtering
282             if filter == "median":
283                 imp = imp.filter(ImageFilter.MedianFilter(size=5))
284 
285             if orig_dim[0] > 100 and orig_dim[1] > 100:
286                 # crop if dimensions higher than 100
287                 imp = image_cropper(imp, [100, 100])
288                 imp = image_resizer(imp, finaldim)
289             else:
290                 imp = image_resizer(imp, [40, 70])
291                 imp = image_resizer(imp, finaldim)
292 
293             #imp.show()  # shows image in external program
```

```
294                 imarray = tomatrix(imp)
295                 dataset[i] = imarray.flatten()
296                 i += 1
297 
298         return dataset
```

A.3 utils.py

```
__author__ = 'gbarbon'

from matplotlib import pyplot as plt
import numpy as np
import random as rnd
import copy as cp


# convert images from 0/1 to -1/1
def image_converter(input_image):
    image = cp.copy(input_image)
    image *= 2
    image -= 1
    return image


# corrupts images
def corrupter(input_image, corr_ratio):
    dim_row = input_image.shape[0]
    dim_col = input_image.shape[1]
    total_dim = dim_col*dim_row
    corrupted_image = cp.copy(input_image).flatten()

    points_to_corr = int((total_dim*corr_ratio)/100)
    for i in range(points_to_corr):
        corr_idx = rnd.randint(0, (dim_row*dim_col-1))
        corrupted_image[corr_idx] *= -1
    corrupted_image.shape = (dim_row, dim_col)

    return corrupted_image


# erase a part of the image starting from the top
def image_eraser(input_image, erase_ratio):
    dim_row = input_image.shape[0]
    dim_col = input_image.shape[1]
    erased_img = cp.copy(input_image).flatten()

    rows_to_erase = int((dim_row*erase_ratio)/100)
    for i in range(dim_col*rows_to_erase):
        erased_img[i] = 1

    erased_img.shape = (dim_row, dim_col)

    return erased_img

#
def semeion_loader(semeion_dir, el):
    data = np.loadtxt(semeion_dir)
    n_el = data.shape[0]
    found = False
    i = rnd.randint(0, n_el)
    while ( not found):
        if data[i][256+el] == 1:
            found = True
            image = cp.copy(data[i])
        else:
```

```
58              i+=1
59          if i>=n_el:
60              i = 0
61      image = np.delete(image, [256,257,258,259,260,261,262,263,264,265,266]) # remove semeion label
62      image = image_converter(image)
63      image = image.reshape(16,16)
64      return image
65
66
67  # Plot and/or save the results
68  def plotter(test_set, result_set, filename, plotbool, savebool):
69      ntest = len(test_set)
70      tickslabels_array_big = ([-0.5, 2.5, 5.5, 8.5], ['0', '3', '6', '9'], [-0.5, 2.5, 5.5, 8.5, 11.5]
71      tickslabels_array_small = ([-0.5, 4.5, 8.5], ['0', '5', '9'], [-0.5, 5.5, 11.5], ['0', '6', '12']
72
73      k = 1
74      if (ntest>5):
75          tickslabels_array = tickslabels_array_small
76          lsize = 6
77      else:
78          tickslabels_array = tickslabels_array_big
79          lsize = 8
80      fig=plt.figure()
81      for i in range(ntest):
82
83          tmp = fig.add_subplot(ntest, 2, k)
84          tmp.imshow(test_set[i], "summer",interpolation="nearest")
85
86          tmp.tick_params(labelsize=lsize)
87          tmp.set_xticks(tickslabels_array[0])
88          tmp.set_xticklabels(tickslabels_array[1])
89          tmp.set_yticks(tickslabels_array[2])
90          tmp.set_yticklabels(tickslabels_array[3])
91          if k==1:
92              #tmp.set_title("Test set")
93              tmp.text(.5, 1.2, 'Test set', horizontalalignment='center', transform=tmp.transAxes, fon
94          k += 1
95
96          tmp = fig.add_subplot(ntest, 2, k)
97          tmp.imshow(result_set[i], "winter", interpolation="nearest")
98          tmp.tick_params(labelsize=lsize)
99          tmp.set_xticks(tickslabels_array[0])
100         tmp.set_xticklabels(tickslabels_array[1])
101         tmp.set_yticks(tickslabels_array[2])
102         tmp.set_yticklabels(tickslabels_array[3])
103         if k==2:
104             tmp.text(.5, 1.2, 'Results', horizontalalignment='center', transform=tmp.transAxes, fonts
105         k += 1
106     fig.subplots_adjust(hspace=.7, wspace=0.01)
107     if plotbool:
108         plt.show()
109     if savebool:
110         fig.savefig(filename, bbox_inches='tight')
```

A.4 tests.py

```
__author__ = 'gbarbon'

import numpy as np
import HopfieldNet
import utils as utl
import imageManager as iM

# Config variables/constant
testnumber = 2
testel = 2  # elements to test
trainel = 2  # elements to train
corr_ratio = 0  # percentage of corruption ratio
erase_ratio = 0 # percentage of image erased
trainers = ["hebbian", "pseudoinv", "storkey"]
trainer = trainers[0]
filetype = "png"
all_trainer = False  # True for all trainers or False for only one
plotbool = False
savebool = True
filters = ["median", "none"]
filter = filters[0]

def test1(trainer_type):
    # corruption_val = 5
    results_dir = "/Users/jian/Dropbox/AI_dropbox/progetto_2014/results/test_1"

    # Create the training patterns
    a_pattern = np.array([[0, 0, 0, 1, 0, 0, 0],
                          [0, 0, 1, 0, 1, 0, 0],
                          [0, 1, 0, 0, 0, 1, 0],
                          [0, 1, 1, 1, 1, 1, 0],
                          [0, 1, 0, 0, 0, 1, 0],
                          [0, 1, 0, 0, 0, 1, 0],
                          [0, 1, 0, 0, 0, 1, 0]])

    b_pattern = np.array([[0, 1, 1, 1, 1, 0, 0],
                          [0, 1, 0, 0, 0, 1, 0],
                          [0, 1, 0, 0, 0, 1, 0],
                          [0, 1, 1, 1, 1, 0, 0],
                          [0, 1, 0, 0, 0, 1, 0],
                          [0, 1, 0, 0, 0, 1, 0],
                          [0, 1, 1, 1, 1, 0, 0]])

    c_pattern = np.array([[0, 1, 1, 1, 1, 1, 0],
                          [0, 1, 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, 1, 0, 0, 0, 0, 0],
                          [0, 1, 1, 1, 1, 1, 0]])

    a_pattern = utl.image_converter(a_pattern)
    b_pattern = utl.image_converter(b_pattern)
    c_pattern = utl.image_converter(c_pattern)

    train_input = np.array([a_pattern.flatten(), b_pattern.flatten(), c_pattern.flatten()])

```

```
58      #hebbian training
59      net = HopfieldNet.HopfieldNet(train_input, trainer_type, [7, 7])
60
61      # creating test set
62      a_test = utl.corrupter(a_pattern, corr_ratio)
63      b_test = utl.corrupter(b_pattern, corr_ratio)
64      c_test = utl.corrupter(c_pattern, corr_ratio)
65
66      # training and testing the net
67      a_result = net.test(a_test)
68      b_result = net.test(b_test)
69      c_result = net.test(c_test)
70
71      #Show the results
72      test_set = np.array([a_test, b_test, c_test])
73      result_set = np.array([a_result, b_result, c_result])
74      utl.plotter(test_set, result_set, results_dir, plotbool, savebool)
75
76
77  def test2(trainer_type, testel, trainel):
78      images_dir = "/Users/jian/Dropbox/AI_dropbox/progetto_2014/dummy_data_set/courier_digits_data_se
79      results_dir = "/Users/jian/Dropbox/AI_dropbox/progetto_2014/results/test_2" + "/" + trainer_type
80      filename = results_dir + "/" + filter + "_" +  "tr" + str(trainel) + "_ts"+ str(testel)
    + "_c" + str(corr_ratio) + "_e" + str(erase_ratio) + "_" + trainer_type + "." + filetype
81      dim = [14, 9]  # in the form rows * cols
82      # testel = 8  # elements for training
83      #corruption_val = 5
84      # trainers = ["hebbian","pseudoinv","storkey"]
85
86      image_dim = [dim[1], dim[0]]  # changing shape for images
87
88      # Loading images data set
89      temp_train = iM.collectimages(image_dim, images_dir, filter)
90
91      # image conversion to 1 and -1 for Hopfield net
92      for i in range(temp_train.shape[0]):
93          temp = utl.image_converter(temp_train[i].reshape(dim))
94          temp_train[i] = temp.flatten()
95
96      train_input = np.zeros((trainel, dim[0] * dim[1]))
97      for i in range(trainel):
98          train_input[i] = temp_train[i]
99
100     # training the net
101     net = HopfieldNet.HopfieldNet(train_input, trainer_type, dim)
102
103     # testing the net
104     test_set = np.zeros((testel, dim[0], dim[1]))
105     result_set = np.zeros((testel, dim[0], dim[1]))
106     for i in range(testel):
107         test_set[i] = temp_train[i].reshape(dim)
108         if corr_ratio != 0:
109             test_set[i] = utl.corrupter(test_set[i], corr_ratio)
110         if erase_ratio != 0:
111             test_set[i] = utl.image_eraser(test_set[i], erase_ratio)
112         result_set[i] = net.test(test_set[i])
113
114     # Plotting and saving results
115     utl.plotter(test_set, result_set, filename, plotbool, savebool)
```

```
116
117
118 def test3(trainer_type, testel, trainel):
119     images_dir = "/Users/jian/Dropbox/AI_dropbox/progetto_2014/dummy_data_set/digital7_digit_data_se
120     results_dir = "/Users/jian/Dropbox/AI_dropbox/progetto_2014/results/test_3" + "/" + trainer_type
121     filename = results_dir + "/" + "tr" + str(trainel) + "_ts"+ str(testel)
    + "_c" + str(corr_ratio) + "_e" + str(erase_ratio) + "_" + trainer_type + "." + filetype
122     dim = [25, 16]  # in the form rows * cols
123     # testel = 5  # elements for training
124     #corruption_val = 10
125
126     image_dim = [dim[1], dim[0]]  # changing shape for images
127
128     # Loading images data set
129     temp_train = iM.collectimages(image_dim, images_dir, filter)
130
131     # image conversion to 1 and -1 for Hopfield net
132     for i in range(temp_train.shape[0]):
133         temp = utl.image_converter(temp_train[i].reshape(dim))
134         temp_train[i] = temp.flatten()
135
136     train_input = np.zeros((trainel, dim[0] * dim[1]))
137     for i in range(trainel):
138         train_input[i] = temp_train[i]
139
140     # training the net
141     net = HopfieldNet.HopfieldNet(train_input, trainer_type, dim)
142
143     # testing the net
144     test_set = np.zeros((testel, dim[0], dim[1]))
145     result_set = np.zeros((testel, dim[0], dim[1]))
146     for i in range(testel):
147         test_set[i] = train_input[i].reshape(dim)
148         if corr_ratio != 0:
149             test_set[i] = utl.corrupter(test_set[i], corr_ratio)
150         if erase_ratio != 0:
151             test_set[i] = utl.image_eraser(test_set[i], erase_ratio)
152         result_set[i] = net.test(test_set[i])
153
154     # Plotting and saving results
155     utl.plotter(test_set, result_set, filename, plotbool, savebool)
156
157 def test_semeion(trainer_type, testel, trainel):
158     images_dir = "/Users/jian/Dropbox/AI_dropbox/progetto_2014/dummy_data_set/courier_digits_data_se
159     results_dir = "/Users/jian/Dropbox/AI_dropbox/progetto_2014/results/test_semeion" + "/" + trainer
160     semeion_dir = "/Users/jian/Dropbox/AI_dropbox/progetto_2014/semeion_data_set/semeion.data"
161     filename = results_dir + "/" + "tr" + str(trainel) + "_ts"+ str(testel)
    + "_c" + str(corr_ratio) + "_e" + str(erase_ratio) + "_" + trainer_type + "." + filetype
162     dim = [16, 16]
163
164     image_dim = [dim[1], dim[0]]  # changing shape for images
165
166     # Loading images data set
167     temp_train = iM.collectimages(image_dim, images_dir, filter)
168
169     # image conversion to 1 and -1 for Hopfield net
170     for i in range(temp_train.shape[0]):
171         temp = utl.image_converter(temp_train[i].reshape(dim))
172         temp_train[i] = temp.flatten()
```

```
173
174     train_input = np.zeros((trainel, dim[0] * dim[1]))
175     for i in range(trainel):
176         train_input[i] = temp_train[i]
177
178     # training the net
179     net = HopfieldNet.HopfieldNet(train_input, trainer_type, dim)
180
181     # loading semeion data set
182     test_set = np.zeros((testel, dim[0], dim[1]))
183     for i in range(testel):
184         test_set[i] = utl.semeion_loader(semeion_dir, i)
185
186     # testing the net
187     result_set = np.zeros((testel, dim[0], dim[1]))
188     for i in range(testel):
189         # test_set[i] = temp_train[i].reshape(dim)
190         if corr_ratio != 0:
191             test_set[i] = utl.corrupter(test_set[i], corr_ratio)
192         if erase_ratio != 0:
193             test_set[i] = utl.image_eraser(test_set[i], erase_ratio)
194         result_set[i] = net.test(test_set[i])
195
196     # Plotting and saving results
197     utl.plotter(test_set, result_set, filename, plotbool, savebool)
198
199
200     # loading semeion data set for training
201
202
203
204 def main():
205     if all_trainer:
206         iterator = trainers
207     else:
208         iterator = [trainer]
209     for i in range(len(iterator)):
210         print("Now trainer is: ", iterator[i])
211         if testnumber == 1:
212             test1(iterator[i])
213         elif testnumber == 2:
214             test2(iterator[i], testel, trainel)
215         elif testnumber == 3:
216             test3(iterator[i], testel, trainel)
217         elif testnumber == 4:
218             test_semeion(iterator[i], testel, trainel)
219     #total2()
220     #filter_hebbian_2()
221
222 def total2():
223     test_couples = [[2,2],[2,3],[3,2],[5,5],[5,10],[8,8],[10,5],[10,10]]
224     for i in range(len(trainers)):
225         for j in range(len(test_couples)):
226             testel = test_couples[j][1]
227             trainel = test_couples[j][0]
228             test2(trainers[i], testel, trainel)
229
230 # remember to set hebbian as trainer before executing
231 def filter_hebbian_2():
```

```
232
233         test_couples = [[2,2],[3,3],[4,4],[5,5],[6,6],[7,7],[8,8],[9,9],[10,10]]
234
235         for j in range(len(test_couples)):
236             testel = test_couples[j][1]
237             trainel = test_couples[j][0]
238             test2(trainer, testel, trainel)
239
240     if __name__ == "__main__":
241         main()
```