Neural Network Computing Project

Liang Niu
N19486574
ln932@nyu.edu

December 6, 2016

1 Program

1.1 Source Code

SOURCE CODE (Python 2.7, numpy needed, anaconda recommended):

```
import numpy as np
from itertools import combinations
import random

MAX_LOOP = 10000 # parameter for AM.predict_converge(): maximum loop times
TEST_LOOP = 100 # parameter for Q2: for a err number, try homw many times
ERR_RANGE = 35   # parameter for Q2: the domain of err number
VERBOSE = False  # True to output more details

def trans(x): # turn {1,0,-1,?} into {#,?, ,_}
    if x == 1: return '#'
    if x == -1: return ' '
    if x == 0: return '?'
    return '_'

def load_data(filename): # load data from txt file
    with open(filename) as f:
        lines = [x.ljust(35, ' ').replace(' ','0').replace('#','1') for x in f.r
        data = [[1 if x=='1' else -1 for x in list(s) ] for s in lines]
        return np.array(data)

def display(data): # visualize a pattern(a 35x1 array)
    assert(data.size and data.size%35==0)
    data.reshape(data.size/35, 35)
    if len(data.shape)==1:
```

```
        data = np.array([data])
    for d in data:
        assert(d.shape[0]==35)
        d = [trans(e) for e in d]
        for i in range(7):
            print ''.join(d[5*i:5*i+5])
        print '-'*80

def disturbe(raw_data, n, mode = "missing"):
    """
    take (data, n, mode) as input,
    data is a (35,) np array,
    generate some noise to data, mode is 'missing' or 'mistake',
    n is the number of noisy point.
    raw data is not changed
    """
    assert(mode == "missing" or mode == "mistake")
    assert(raw_data.size == 35)
    assert(n<=35 and n>=0)
    factor = 0 if mode == "missing" else -1
    data = np.copy(raw_data)
    data = data.reshape((35,))
    chose = random.sample(range(35), n)
    for ind in chose:
        data[ind] *= factor
    return data

class AM:
    def __init__(self, X):
        self.q = np.size(X,0)
        self.n = np.size(X,1)
        self.w = np.zeros((self.n, self.n))
        self.threshold = 0
        self.w = X.T.dot(X)
        self.w0 = np.copy(self.w)
        np.fill_diagonal(self.w0, 0)
        # print "w is:\n",self.w
        # print "w0 is:\n",self.w0
        def f_notvec(x):
            if x > self.threshold:
                return 1
            elif x < self.threshold:
                return -1
            else:
                return 0
        self.f = np.vectorize(f_notvec,cache=True)
```

```
    def predict(self, x):
        """
        simply predict
        """
        return (self.f(x.dot(self.w)), self.f(x.dot(self.w0)))

    def predict_converge(self, x):
        """
        predict iteratively until converge
        """
        p0 = self.f(x.dot(self.w0))
        for i in range(MAX_LOOP): # at most iterate MAX_LOOP times
            p0_temp = self.f(p0.dot(self.w0))
            if np.array_equal(p0, p0_temp):
                break
            else:
                p0 = p0_temp
        p = self.f(x.dot(self.w))
        for i in range(MAX_LOOP): # at most iterate MAX_LOOP times
            p_temp = self.f(p.dot(self.w))
            if np.array_equal(p, p_temp):
                break
            else:
                p = p_temp
        return (p, p0)

if __name__ == "__main__":
    # load data
    train_file = "TenDigitPatterns.txt"
    train_data = load_data(train_file)
    if VERBOSE:
        display(train_data)

    ###################
    # first question #
    ###################
    good_c = []
    if True:
    # if False:
        print "*"*80
        print "# first question #"
        for num in range(1,8):
            for c in combinations(range(10), num):
                # print c
                part_train_data = train_data[np.array(c)]
```

```
                am = AM(part_train_data)

                for p in [(0,am.predict_converge(part_train_data)), (1,am.predic
                    mode = "converge" if p[0]==0 else "direct"
                    pre = p[1]
                    p0, p1= False, False
                    if np.array_equal(pre[0], part_train_data):
                        p0 = True
                    if np.array_equal(pre[1], part_train_data):
                        p1 = True
                    if p0 or p1:
                        print mode, c,
                        good_c.append(c)
                        if p0:
                            print "p",
                        if p1:
                            print "p0",
                        print ""
    print good_c

####################
# second question #
####################
if True:
# if False:
    print "*"*80
    print "# second question #"
    for num in range(1,8):
        for c in combinations(range(10), num):
            if c in good_c:
                print c
                part_train_data = train_data[np.array(c)]
                am = AM(part_train_data)

                D_N = part_train_data.shape[0]
                for err_n in range(ERR_RANGE): # error number
                    err_missing = 0.0
                    err_mistake = 0.0
                    for i in range(TEST_LOOP):
                        for d in part_train_data:
                            missing_d= disturbe(d, err_n, mode="missing")
                            mistake_d= disturbe(d, err_n, mode="mistake")
                            missing_pre = am.predict(missing_d)[1] # using u
                            mistake_pre = am.predict(mistake_d)[1] # using u
                            if VERBOSE:
                                display(d)
```

```
                                display(missing_d)
                                display(mistake_d)
                                display(missing_pre)
                                display(mistake_pre)
                            if not np.array_equal(missing_pre, d):
                                err_missing += 1
                            if not np.array_equal(mistake_pre, d):
                                err_mistake += 1
                    err_rate_missing = err_missing/(TEST_LOOP * D_N)
                    err_rate_mistake = err_mistake/(TEST_LOOP * D_N)
                    print "When err is", err_n, ", missing ERR rate:", err_ra

##################
# Third question #
##################
if True:
# if False:
    print "*"*80
    print "# second question #"
    def spurious(pre, train_data): # given a predict, judge if it's in the s
        for d in train_data:
            if np.array_equal(d, pre):
                return False
        return True
    # c= (1, 4, 6, 7, 9)
    # part_train_data = train_data[np.array(c)]
    # am = AM(part_train_data)
    set_p = set()
    total_n = 0

    # find out the spurious patterns
    for num in range(1,8):
        for c in combinations(range(10), num):
            if c in good_c:
                # print c
                part_train_data = train_data[np.array(c)]
                am = AM(part_train_data)
                # pre = am.predict_converge(part_train_data)[0]
                pre = am.predict(part_train_data)[1] # using w0 to predict
                total_n += pre.shape[0]
                for p in pre:
                    p.reshape(35,)
                    if spurious(p, part_train_data):
                        t = tuple(p)
                        set_p.add(t)
```

```
        # show these spurious patterns
        for sp in set_p:
            display(np.array(sp))
        print len(set_p)
        print total_n

print "*"*80
```

1.2 Some Explaination

Github https://github.com/wangxiaodiu/NN-project
please check this github repo to check full output and everything.

Run Environment I also included a source file of my program in the zipped file. If you want to run it, please make sure that you have numpy installed. I am using Python 2.7 in this project. Also Anaconda is highly recommended.

Parameters There are some parameters in this program that one can adjust to see the result. I put them on the begining of the source code and make sure they are well commented so that you can know the meaning for every parameter.

Toggle There are three main questions so there are three corresponding parts in the "main" part of the program. And before every part, there is a "if" statement to let you decide whether you want to run this part. Every part is indepent with each other. By default, they will all run.

Run Time On my machine(Macbook Pro 13, early 2015), the prgram need about half a minute to finish. This may be different from 10s to 5m I guess.

Others Because the output will be a lot, I suggest to use the following command to redirect the output:

```
python am.py | tee output.txt
```

2 Analysis and Answers

2.1 Program

I wrote a class called AM to do the job. It can predict in two policies: direct predict and predict until we get to a converge or run out of loop times. Also you can choose to use w or w0, w is the weight matrix that diagonal elements all untouched, w0 is the weight matrix that diagonal elements are all set to 0. Which means I implemented 4 kinds of predict methods:(direct, w), (direct,

w0), (converge, w), (converge, w0). You can see in the Q1 part code, I tried all of 4 kinds of prediction.
display() can help to display the patterns for you to recognize.
disturbe() is to help make noise into the data.

2.2 Q1

The output for Q1 is in a manner that "method,numbers,weight".
"method" maybe direct or converge. Direct means predict just use $w * input$, "converge" means predict iteratively until converge or reach the maximum loop times.
"numbers" is the working combination of the patterns.
"weight" may be "p" or "p0". "p" means we get this result by using the weight matrix with digonal elements stay untouched. "p0" means we get this result by using the weght matrix with all diagonal elemets set 0.
As we can see, there are 456 kinds of combinations that can be stored and retrived smoothly. Thus the maximum capacity of AM in this problem is 5.
The full results of combinations that can be stored is attached.
Also, by analysing the patterns that can be combined and stored, we can see that most of them are "more orthogonal" than those cannot. For example, $(0, 1, 2, 3, 4)$ and $(0, 1, 2, 4, 6)$can be stored, but $(0, 1, 2, 4, 5)$ cannot be stored, by looking into the patterns, we can see that 5 is intersected a lot with 0, 2, 3, especially the first 6 points and the last 6 points. Also, we can notice that number 4's pattern is not orthogonal with most patterns, which is corresponded to that 4 appears a lot in the combinations that can be stored, becasue the network don't need too much "capacity" for number 4 to store.
So the conclusion is that more orthogonal, better to store in network.

2.3 Q2

For this question, I just do the experiments under those patterns that can be stored. And I implemented two kinds of noise: the first one is that missing, which turn some points in the pattern into 0, the second one is mistake, which is to flip some points in the pattern.
Observe the result, we can conclude that:
1. When number of errors increased, error rate increased.
2. When number of patterns stored increased, the number of errors that it can tolerate decreased.
3. When error number is big enough, the network will only give out wrong result.
4. Normally, missing error rate is lower that mistake error rate.
Here is a part of the output, just to demonstrate some of the conclusion:

```
(0,)
When err is 0 , missing ERR rate: 0.0 , mistake ERR rate: 0.0
When err is 1 , missing ERR rate: 0.0 , mistake ERR rate: 0.0
```

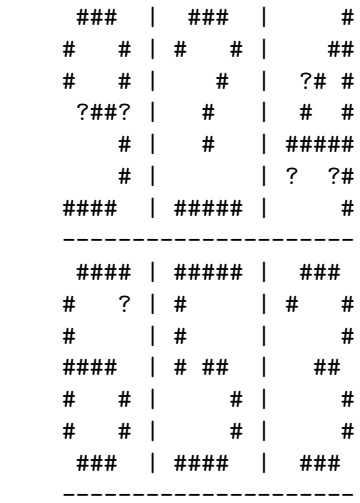
```
When err is 2 , missing ERR rate: 0.0 , mistake ERR rate: 0.0
When err is 3 , missing ERR rate: 0.0 , mistake ERR rate: 0.0
When err is 4 , missing ERR rate: 0.0 , mistake ERR rate: 0.0
When err is 5 , missing ERR rate: 0.0 , mistake ERR rate: 0.0
When err is 6 , missing ERR rate: 0.0 , mistake ERR rate: 0.0
When err is 7 , missing ERR rate: 0.0 , mistake ERR rate: 0.0
When err is 8 , missing ERR rate: 0.0 , mistake ERR rate: 0.0
When err is 9 , missing ERR rate: 0.0 , mistake ERR rate: 0.0
When err is 10 , missing ERR rate: 0.0 , mistake ERR rate: 0.0
When err is 11 , missing ERR rate: 0.0 , mistake ERR rate: 0.0
When err is 12 , missing ERR rate: 0.0 , mistake ERR rate: 0.0
When err is 13 , missing ERR rate: 0.0 , mistake ERR rate: 0.0
When err is 14 , missing ERR rate: 0.0 , mistake ERR rate: 0.0
When err is 15 , missing ERR rate: 0.0 , mistake ERR rate: 0.0
When err is 16 , missing ERR rate: 0.0 , mistake ERR rate: 0.0
When err is 17 , missing ERR rate: 0.0 , mistake ERR rate: 1.0
When err is 18 , missing ERR rate: 0.0 , mistake ERR rate: 1.0
When err is 19 , missing ERR rate: 0.0 , mistake ERR rate: 1.0
When err is 20 , missing ERR rate: 0.0 , mistake ERR rate: 1.0
When err is 21 , missing ERR rate: 0.0 , mistake ERR rate: 1.0
When err is 22 , missing ERR rate: 0.0 , mistake ERR rate: 1.0
When err is 23 , missing ERR rate: 0.0 , mistake ERR rate: 1.0
When err is 24 , missing ERR rate: 0.0 , mistake ERR rate: 1.0
When err is 25 , missing ERR rate: 0.0 , mistake ERR rate: 1.0
When err is 26 , missing ERR rate: 0.0 , mistake ERR rate: 1.0
When err is 27 , missing ERR rate: 0.0 , mistake ERR rate: 1.0
When err is 28 , missing ERR rate: 0.0 , mistake ERR rate: 1.0
When err is 29 , missing ERR rate: 0.0 , mistake ERR rate: 1.0
When err is 30 , missing ERR rate: 0.0 , mistake ERR rate: 1.0
When err is 31 , missing ERR rate: 0.0 , mistake ERR rate: 1.0
When err is 32 , missing ERR rate: 0.0 , mistake ERR rate: 1.0
When err is 33 , missing ERR rate: 0.0 , mistake ERR rate: 1.0
When err is 34 , missing ERR rate: 1.0 , mistake ERR rate: 1.0
(1, 3, 9)
When err is 0 , missing ERR rate: 0.0 , mistake ERR rate: 0.0
When err is 1 , missing ERR rate: 0.0 , mistake ERR rate: 0.0633333333333
When err is 2 , missing ERR rate: 0.02 , mistake ERR rate: 0.19
When err is 3 , missing ERR rate: 0.0233333333333 , mistake ERR rate: 0.173333333333
When err is 4 , missing ERR rate: 0.07 , mistake ERR rate: 0.3
When err is 5 , missing ERR rate: 0.0433333333333 , mistake ERR rate: 0.31
When err is 6 , missing ERR rate: 0.14 , mistake ERR rate: 0.423333333333
When err is 7 , missing ERR rate: 0.09 , mistake ERR rate: 0.47
When err is 8 , missing ERR rate: 0.146666666667 , mistake ERR rate: 0.5
When err is 9 , missing ERR rate: 0.14 , mistake ERR rate: 0.526666666667
When err is 10 , missing ERR rate: 0.233333333333 , mistake ERR rate: 0.62
When err is 11 , missing ERR rate: 0.173333333333 , mistake ERR rate: 0.636666666667
```

```
When err is 12 , missing ERR rate: 0.28 , mistake ERR rate: 0.75
When err is 13 , missing ERR rate: 0.23 , mistake ERR rate: 0.823333333333
When err is 14 , missing ERR rate: 0.34 , mistake ERR rate: 0.93
When err is 15 , missing ERR rate: 0.296666666667 , mistake ERR rate: 0.98
When err is 16 , missing ERR rate: 0.376666666667 , mistake ERR rate: 1.0
When err is 17 , missing ERR rate: 0.34 , mistake ERR rate: 1.0
When err is 18 , missing ERR rate: 0.453333333333 , mistake ERR rate: 1.0
When err is 19 , missing ERR rate: 0.393333333333 , mistake ERR rate: 1.0
When err is 20 , missing ERR rate: 0.48 , mistake ERR rate: 1.0
When err is 21 , missing ERR rate: 0.4 , mistake ERR rate: 1.0
When err is 22 , missing ERR rate: 0.556666666667 , mistake ERR rate: 1.0
When err is 23 , missing ERR rate: 0.493333333333 , mistake ERR rate: 1.0
When err is 24 , missing ERR rate: 0.62 , mistake ERR rate: 1.0
When err is 25 , missing ERR rate: 0.593333333333 , mistake ERR rate: 1.0
When err is 26 , missing ERR rate: 0.66 , mistake ERR rate: 1.0
When err is 27 , missing ERR rate: 0.63 , mistake ERR rate: 1.0
When err is 28 , missing ERR rate: 0.763333333333 , mistake ERR rate: 1.0
When err is 29 , missing ERR rate: 0.756666666667 , mistake ERR rate: 1.0
When err is 30 , missing ERR rate: 0.94 , mistake ERR rate: 1.0
When err is 31 , missing ERR rate: 0.896666666667 , mistake ERR rate: 1.0
When err is 32 , missing ERR rate: 1.0 , mistake ERR rate: 1.0
When err is 33 , missing ERR rate: 1.0 , mistake ERR rate: 1.0
When err is 34 , missing ERR rate: 1.0 , mistake ERR rate: 1.0
(1, 4, 6, 7, 9)
When err is 0 , missing ERR rate: 0.0 , mistake ERR rate: 0.0
When err is 1 , missing ERR rate: 0.08 , mistake ERR rate: 0.2
When err is 2 , missing ERR rate: 0.168 , mistake ERR rate: 0.2
When err is 3 , missing ERR rate: 0.138 , mistake ERR rate: 0.32
When err is 4 , missing ERR rate: 0.18 , mistake ERR rate: 0.356
When err is 5 , missing ERR rate: 0.152 , mistake ERR rate: 0.49
When err is 6 , missing ERR rate: 0.208 , mistake ERR rate: 0.542
When err is 7 , missing ERR rate: 0.19 , mistake ERR rate: 0.654
When err is 8 , missing ERR rate: 0.27 , mistake ERR rate: 0.754
When err is 9 , missing ERR rate: 0.252 , mistake ERR rate: 0.856
When err is 10 , missing ERR rate: 0.312 , mistake ERR rate: 0.914
When err is 11 , missing ERR rate: 0.266 , mistake ERR rate: 0.968
When err is 12 , missing ERR rate: 0.362 , mistake ERR rate: 0.99
When err is 13 , missing ERR rate: 0.33 , mistake ERR rate: 1.0
When err is 14 , missing ERR rate: 0.426 , mistake ERR rate: 1.0
When err is 15 , missing ERR rate: 0.398 , mistake ERR rate: 1.0
When err is 16 , missing ERR rate: 0.476 , mistake ERR rate: 1.0
When err is 17 , missing ERR rate: 0.436 , mistake ERR rate: 1.0
When err is 18 , missing ERR rate: 0.57 , mistake ERR rate: 1.0
When err is 19 , missing ERR rate: 0.518 , mistake ERR rate: 1.0
When err is 20 , missing ERR rate: 0.638 , mistake ERR rate: 1.0
When err is 21 , missing ERR rate: 0.57 , mistake ERR rate: 1.0
```

```
When err is 22 , missing ERR rate: 0.692 , mistake ERR rate: 1.0
When err is 23 , missing ERR rate: 0.71 , mistake ERR rate: 1.0
When err is 24 , missing ERR rate: 0.81 , mistake ERR rate: 1.0
When err is 25 , missing ERR rate: 0.756 , mistake ERR rate: 1.0
When err is 26 , missing ERR rate: 0.906 , mistake ERR rate: 1.0
When err is 27 , missing ERR rate: 0.88 , mistake ERR rate: 1.0
When err is 28 , missing ERR rate: 0.978 , mistake ERR rate: 1.0
When err is 29 , missing ERR rate: 0.958 , mistake ERR rate: 1.0
When err is 30 , missing ERR rate: 0.998 , mistake ERR rate: 1.0
When err is 31 , missing ERR rate: 0.994 , mistake ERR rate: 1.0
When err is 32 , missing ERR rate: 1.0 , mistake ERR rate: 1.0
When err is 33 , missing ERR rate: 1.0 , mistake ERR rate: 1.0
When err is 34 , missing ERR rate: 1.0 , mistake ERR rate: 1.0
```

2.4 Q3

For this question, I just do the experiments under those patterns that can be stored. And I implemented using direct predict with w0. Typically, the spurious patterns looks like some of the train patterns very much, just with some points flipped or missing. For example, there are some spurious patterns(1 is #, -1 is space, 0 is ?) in the following figure.



In the number 9, some points are missing. In the number 2, some points are flipped(from 1 to -1), in the number 4, some points which should be -1 are missing.