POSTER: COMPUTATION TECHNIQUES FOR ENCRYPTED DATA

Gajendra Deshpande
KLS Gogte Institute of Technology, India

https://gcdeshpande.github.io

Contents

- Homomorphic Encryption
- Properties of Homomorphic Encryption
- Summary of Homomorphic Properties
- The Example
- Experiment with RSA Algorithm
- Results
- Machine Learning and Homomorphic Encryption
- Conclusion

Homomorphic Encryption

- Several billion devices are currently connected to the Internet, and this number will continue to grow.
- This is a consequence of not only more people becoming interested in consumer electronics but also more sensors and actuators being incorporated into everyday electronics, household appliances, and the general infrastructure.
- Since most of these devices are not able to process data locally, they will often upload it to a third party for processing.
- However, this data may be private, the third party may not be trustworthy, or both. Therefore, the data should be encrypted before it is transferred

Homomorphic Encryption

- Imagine taking all of your credit card statements and locking them into a safe, to which you have the only key. Your statements are now protected from prying eyes. This is what encryption does.
- But what if you wanted to analyse your expenditure on groceries in the last 12 months? First you would have to unlock the safe and retrieve the statements. So now the documents are out in the open and they can be read by anyone. This is what decryption does.
- The difference with Homomorphic Encryption is that you can create your report without taking the documents out of the safe.

Properties of Homomorphic Encryption

- **Additive Homomorphic Encryption:**

A Homomorphic encryption is additive, if

$Ek\ (PT1 \oplus PT2) = Ek\ (PT1) \oplus Ek\ (PT2)$

As the encryption function is additively homomorphic, the following identities can be described:

The product of two cipher texts will decrypt to the sum of their corresponding plaintexts,

$D\ (E\ (m1) \cdot E\ (m2) \bmod n) = m1 + m2 \bmod n.$

The product of a cipher text with a plaintext raising g will decrypt to the sum of the corresponding plaintexts,

$D\ (E\ (m1) \cdot g^{m2} \bmod n2) = m1 + m2 \bmod n.$

Properties of Homomorphic Encryption

- **Multiplicative Homomorphic Encryption:** Homomorphic encryption is multiplicative, if

$Ek\ (PT1 \otimes PT2) = Ek\ (PT1) \otimes Ek\ (PT2)$

- The homomorphic property of the RSA.

Suppose there are two cipher texts, CT1 and CT2.

$CT1 = m1^{e} \bmod n$

$CT2 = m2^{e} \bmod n$

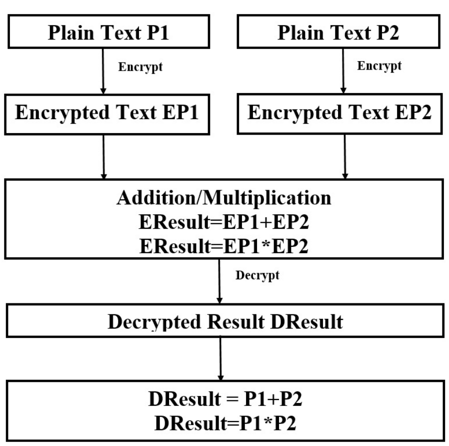
$CT1 \cdot CT2 = m1^{e} \cdot m2^{e} \bmod n$

So, multiplicative property: $(m1 \cdot m2)^{e} \bmod n$

Summary of Homomorphic Properties

Algorithm	Additive	Multiplicative	Applications
RSA	No.	Yes	To secure Internet Banking and credit card transactions
Paillier	Yes	No	E-voting system
ElGamal	No.	Yes	In Hybrid Systems

The Example



- P1=5; P2=10
- EP1=50 EP2=100
- EResult1=50+100=150; Eresult2=50*100=5000
- DResult1=15; DResult2=50

Experiment with RSA Algorithm

- Selecting two large primes at random: p, q
- Computing their system modulus $n=p.q$
- Note $ø(n)=(p-1)(q-1)$
- Selecting at random the encryption key e where $1<e<ø(n)$, $gcd(e,ø(n))=1$
- Solve following equation to find decryption key d
- $e.d=1 \bmod ø(n)$ and $0 \le d \le n$
- Publish their public encryption key: pu={e,n}
- Keep secret private decryption key: pr={d,n}

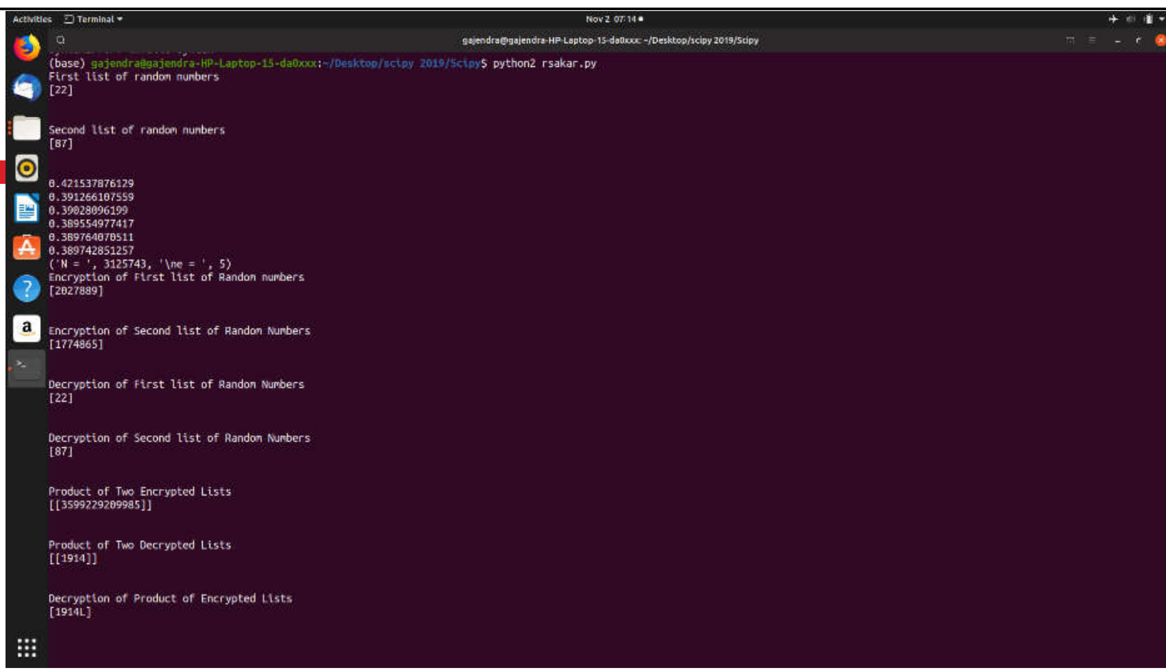
Experiment with RSA Algorithm

- To encrypt a message M the sender:
  - obtains public key of recipient pu={e,n}
  - computes: $C = m^e \bmod n$, where $0 \le m < n$
- To decrypt the ciphertext C the owner:
  - uses their private key pr={d,n}
  - computes: $M = c^d \bmod n$

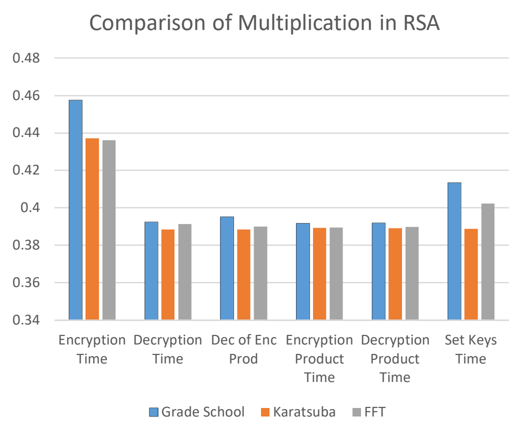
Experiment with RSA Algorithm

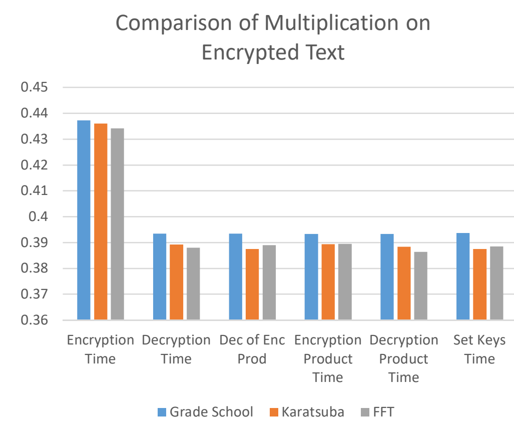
Grade School Method	Karatsuba Method	Fast Fourier Transform
$O(n^2)$	$O(n^{\log_2 3}) = O(n^{1.58})$	$\Theta(n \log(n) \log(\log(n)))$

- The multiplication algorithms were implemented in two steps
- Multiplication algorithms in RSA algorithm were replaced by Karatsuba and FFT methods one after another.
- Multiplication operations were performed on encrypted numbers by Karatsuba and FFT Methods.

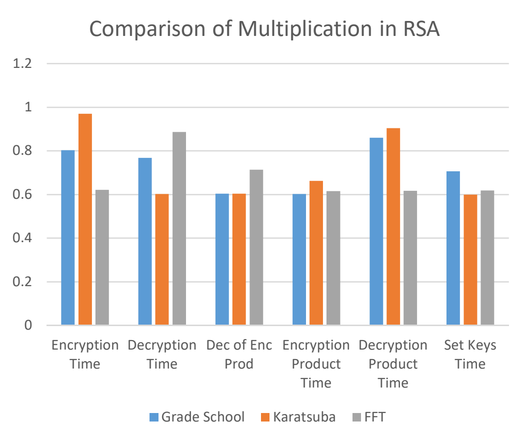


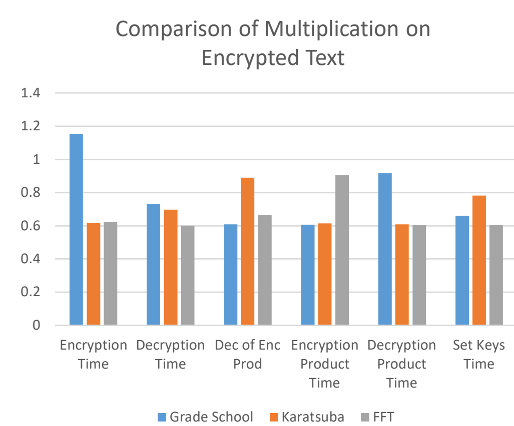
Results-High Resource





Results-Low Resource

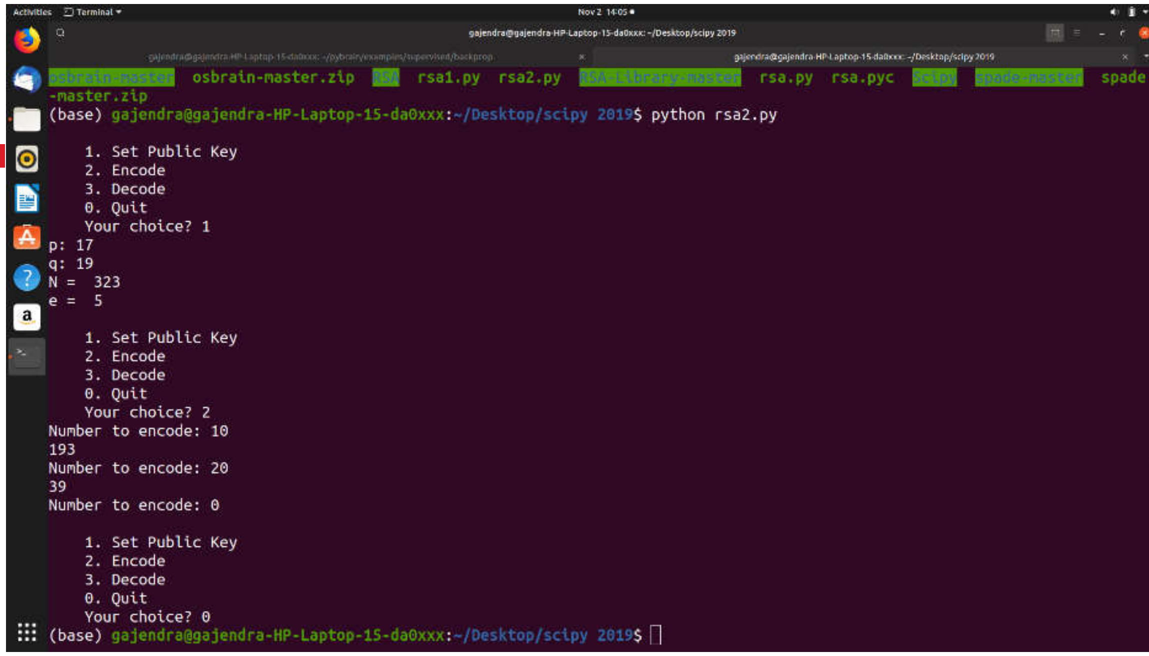


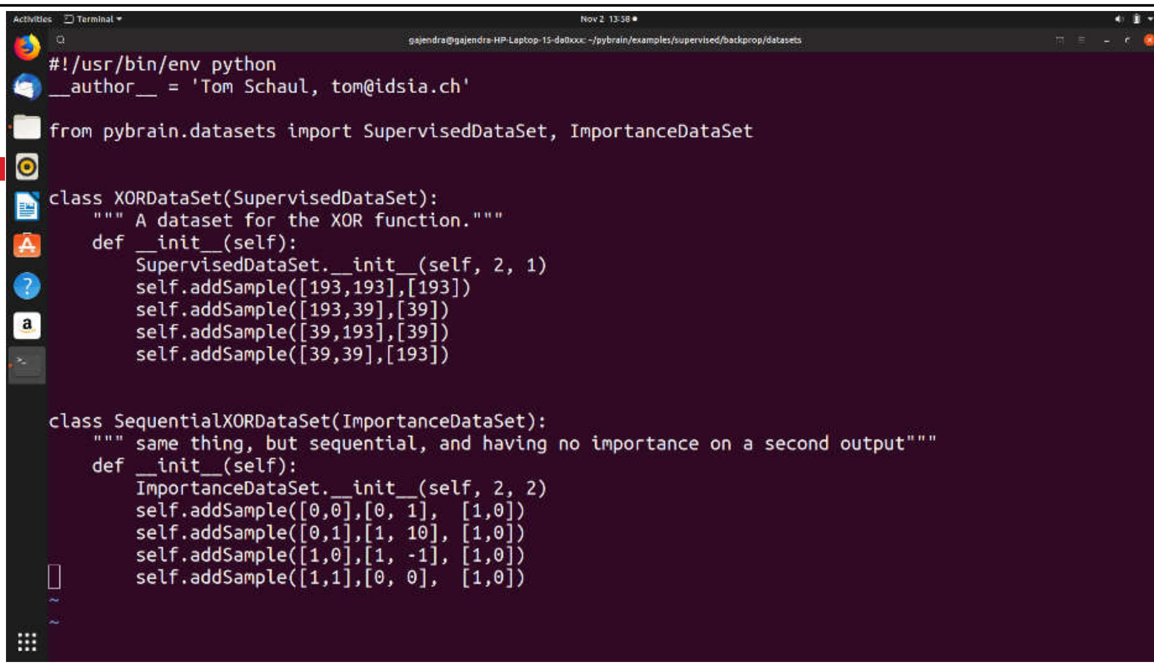


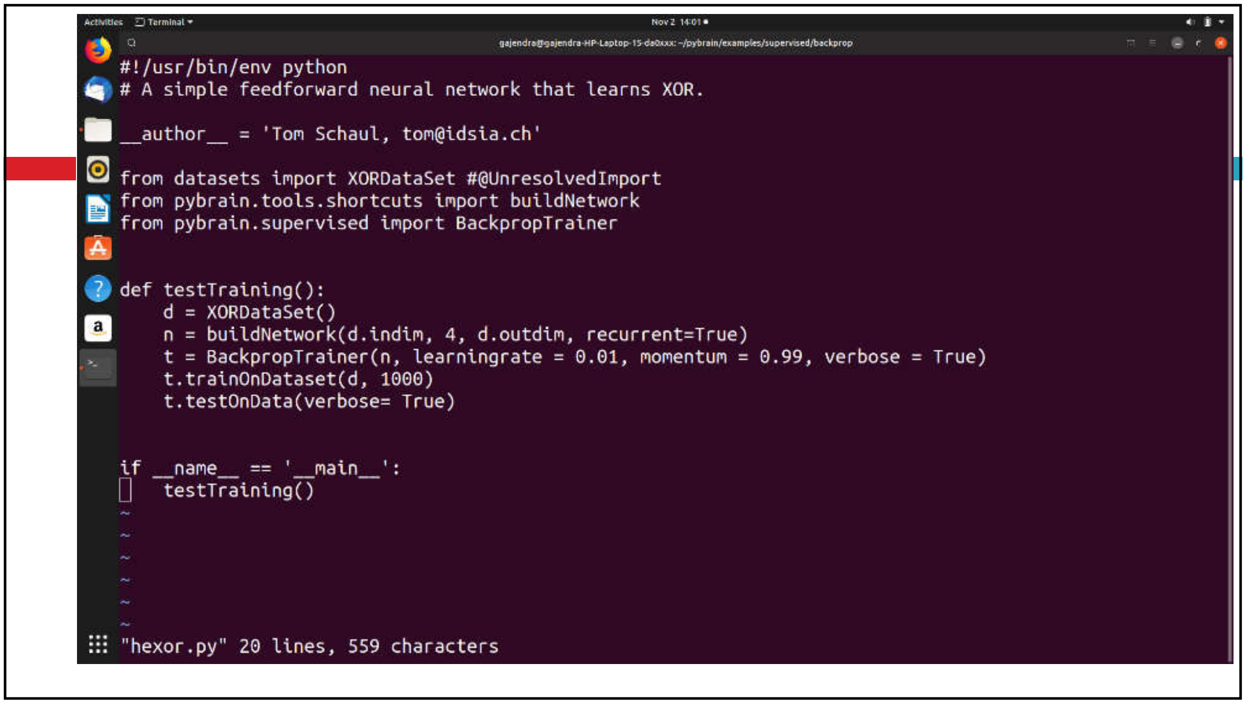
Machine Learning and Homomorphic Encryption

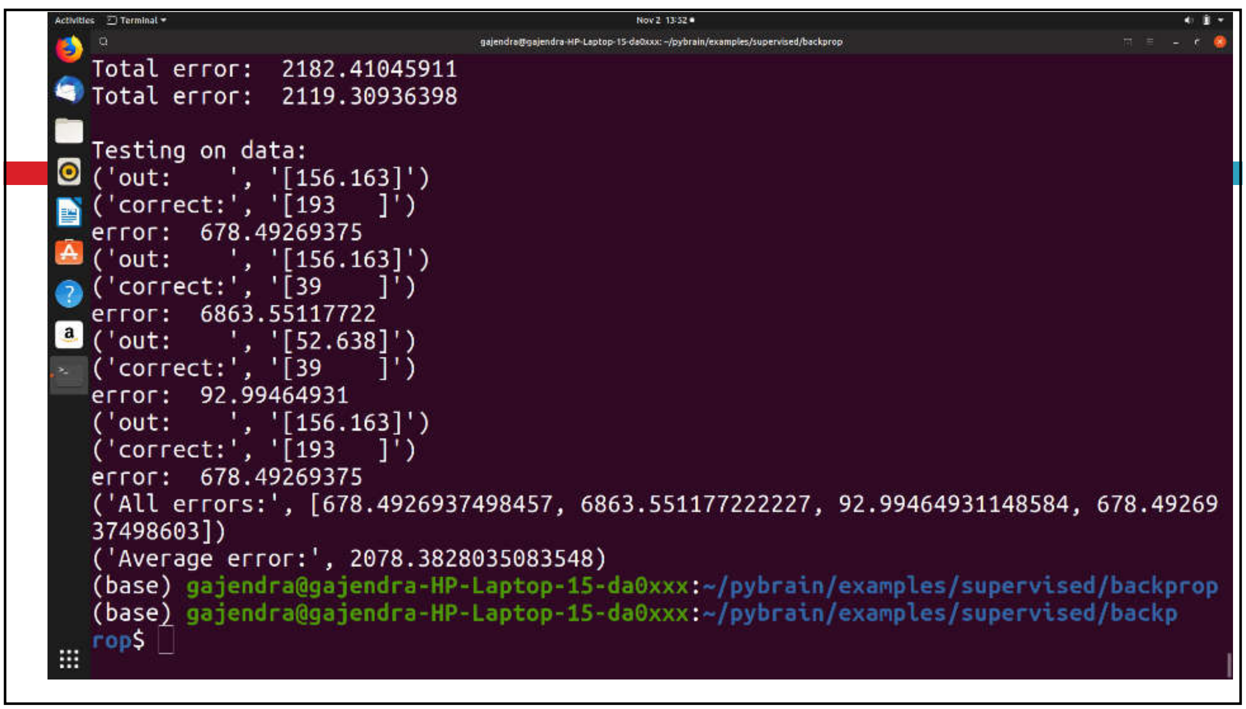
- Pre-processing: Map the numbers in the dataset to random numbers
- Encrypt the data set using cryptographic algorithms such as RSA, paillier or any other cryptosystem
- Perform the computations on encrypted data
- Decrypt the results
- Post-processing: rounding up/down, remap random numbers to original numbers

```
#!/usr/bin/env python
__author__ = 'Tom Schaul, tom@idsia.ch'

from pybrain.datasets import SupervisedDataSet, ImportanceDataSet


class XORDataSet(SupervisedDataSet):
    """ A dataset for the XOR function."""
    def __init__(self):
        SupervisedDataSet.__init__(self, 2, 1)
        self.addSample([0,0],[0])
        self.addSample([0,1],[1])
        self.addSample([1,0],[1])
        self.addSample([1,1],[0])


class SequentialXORDataSet(ImportanceDataSet):
    """ same thing, but sequential, and having no importance on a second output"""
    def __init__(self):
        ImportanceDataSet.__init__(self, 2, 2)
        self.addSample([0,0],[0, 1],  [1,0])
        self.addSample([0,1],[1, 10], [1,0])
        self.addSample([1,0],[1, -1], [1,0])
        self.addSample([1,1],[0, 0],  [1,0])
~
~
"xorco.py" 24 lines, 801 characters
```


osbrain-master osbrain-master.zip RSA rsa1.py rsa2.py RSA-Library-master rsa.py rsa.pyc Scipy spade-master spade
-master.zip
(base) gajendra@gajendra-HP-Laptop-15-da0xxx:~/Desktop/scipy 2019$ python rsa2.py
1. Set Public Key
2. Encode
3. Decode
0. Quit
Your choice? 1
p: 17
q: 19
N = 323
e = 5
1. Set Public Key
2. Encode
3. Decode
0. Quit
Your choice? 2
Number to encode: 10
193
Number to encode: 20
39
Number to encode: 0
1. Set Public Key
2. Encode
3. Decode
0. Quit
Your choice? 0
(base) gajendra@gajendra-HP-Laptop-15-da0xxx:~/Desktop/scipy 2019$


#!/usr/bin/env python
__author__ = 'Tom Schaul, tom@idsia.ch'
from pybrain.datasets import SupervisedDataSet, ImportanceDataSet
class XORDataSet(SupervisedDataSet):
""" A dataset for the XOR function."""
def __init__(self):
SupervisedDataSet.__init__(self, 2, 1)
self.addSample([193,193],[193])
self.addSample([193,39],[39])
self.addSample([39,193],[39])
self.addSample([39,39],[193])
class SequentialXORDataSet(ImportanceDataSet):
""" same thing, but sequential, and having no importance on a second output"""
def __init__(self):
ImportanceDataSet.__init__(self, 2, 2)
self.addSample([0,0],[0, 1], [1,0])
self.addSample([0,1],[1, 10], [1,0])
self.addSample([1,0],[1, -1], [1,0])
self.addSample([1,1],[0, 0], [1,0])


#!/usr/bin/env python
# A simple feedforward neural network that learns XOR.
__author__ = 'Tom Schaul, tom@idsia.ch'
from datasets import XORDataSet #@UnresolvedImport
from pybrain.tools.shortcuts import buildNetwork
from pybrain.supervised import BackpropTrainer
def testTraining():
d = XORDataSet()
n = buildNetwork(d.indim, 4, d.outdim, recurrent=True)
t = BackpropTrainer(n, learningrate = 0.01, momentum = 0.99, verbose = True)
t.trainOnDataset(d, 1000)
t.testOnData(verbose= True)
if __name__ == '__main__':
testTraining()
"hexor.py" 20 lines, 559 characters


Total error: 2182.41045911
Total error: 2119.30936398
Testing on data:
('out: ', '[156.163]')
('correct:', '[193]')
error: 678.49269375
('out: ', '[156.163]')
('correct:', '[39]')
error: 6863.55117722
('out: ', '[52.638]')
('correct:', '[39]')
error: 92.99464931
('out: ', '[156.163]')
('correct:', '[193]')
error: 678.49269375
('All errors:', [678.4926937498457, 6863.551177222227, 92.99464931148584, 678.4926937498603])
('Average error:', 2078.3828035083548)
(base) gajendra@gajendra-HP-Laptop-15-da0xxx:~/pybrain/examples/supervised/backprop
(base) gajendra@gajendra-HP-Laptop-15-da0xxx:~/pybrain/examples/supervised/backprop$

Conclusion

- Homomorphic Encryption enables computation on untrusted resource. The Computation time over cipher text can be reduced by using Karatsuba or FFT techniques.
- Training and testing machine learning model may involve additional steps such as pre-processing and post-processing and results into additional computational complexity.