Reservoir Computer

Navid Mousavi

April 2021

Here I have described my I implemented reservoir computing network to predict the Lorenz system time series up to a few Lyapunov time. The time series in question can be generated by numerical integration of the Lorenz equations,

$$\dot{x} = -\sigma x + \sigma y$$
$$\dot{y} = -xz + rx - y$$
$$\dot{z} = xy - bz.$$

These equations describe a chaotic system, which means it becomes unpredictable after Lyapunov time. However, the reservoir computer does a descent job in predicting chaotic time series [].

Reservoir computer is a black box of neurons with random connections to each other, and obeys the following dynamics[],

$$r_i(t+1) = g(\sum_j W_{ij} r_j(t) + \sum_k^N W_{ik}^{in} x_k)$$
$$O_i(t+1) = \sum_j^M W_{ij}^{out} r_j(t+1)$$

$\mathbb{W}$, and $\mathbb{W}^{in}$ are the weights connecting reservoir neurons to each other and input to the reservoir neurons respectively. They should be chosen randomly in the beginning and remain constant. $\mathbb{W}^{out}$ is the weight connecting reservoir to outputs and is the only trainable weight in the reservoir computer implemented here. r_i represents the value of neuron i in the reservoir

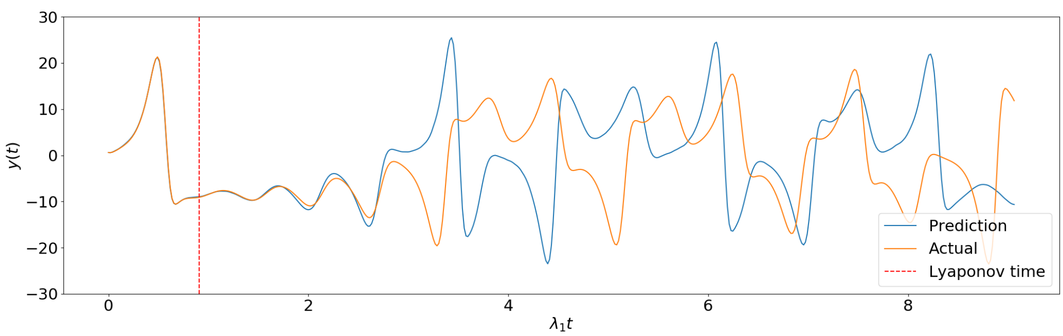


Figure 1: Comparison of the prediction with the actual data for component y of the Lorenz equation.

Training	Neurons	Layers	Spectral radius	Sparsity	Lyapunov exponent
1 (setup in Fig(1))	1000	1	1.2	0.1	-0.08
2	300	1	1000	0.01	4.8
3	300	1	0.01	0.01	-4.8

Table 1: Caption

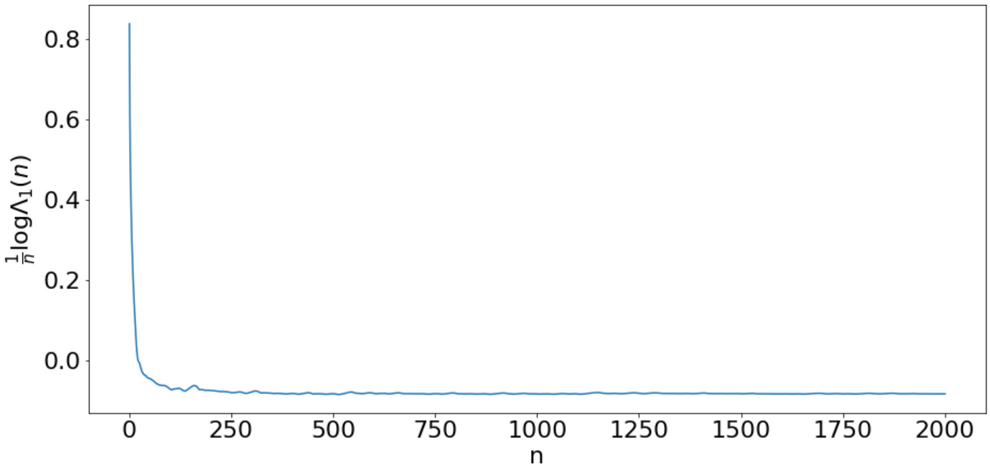
which is referred to as reservoir state in the literature, and g is a nonlinear activation function. Here tanh() is used as the nonlinear activation function. Both $\mathbb{W}$ and $\mathbb{W}^{in}$ components are chosen from uniform distribution in the interval $[-1, 1]$, and $[-0.1, 0.1]$ respectively. The reservoir state can be initialized with zeros. In order to train the output weights one can use back propagation like a normal feed forward neural network with minimizing the loss function (energy) defined as $H = \frac{1}{2}\sum_t |T_t - O_t|^2$ or simply by saving the states of the reservoir through the input feeding time and performing a least squares regression at the end to find the output weights $\mathbb{W}^{out}$. Here both methods are tried and the results presented are generated using weights calculated by ridge regression with penalty parameter 0.1.

Time series were generated by integrating Lorenz equation with $\sigma = 10, r = 28,\ b = \frac{8}{3}$, during $t \in [0, 50]$ with time step size of $dt = 0.02$. Training set consists first 80% of the data and the remaining 20% is used as test set. Fig.(1) shows the comparison of the generated prediction against the actual data. It can be seen that the reservoir successfully has predicted the chaotic time series for more around Lyapunov time scales. The result is produced by a reservoir having 1000 neurons, 1 layer, with spectral radius of 1.2, where neurons are connected with probability 0.1.

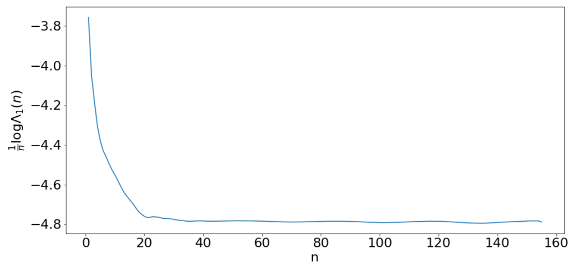
The result is very much dependent on the architecture of the reservoir which is a random outcome. Using the same parameters, with different random seeds will yield different results. This might be due to how the reservoir neurons are connected to each other and determines how well the dynamics of the reservoir can reconstruct the time series fed.

Different degrees of Sparsity were tested and for the best setup that I found the neurons in reservoir were connected with a probability of 0.1. I tried Erdös-Renyi random graph also with degree $d = 6$ which assumes there is a connection between two nodes with probability p, having $np = d$. This means for 1000 neurons setup there must be a connection in adjacency matrix with $p = 0.006$ and $p = 0.02$ for 300 neurons, but I did not manage to find a good performance using Erdös-Renyi random graph.

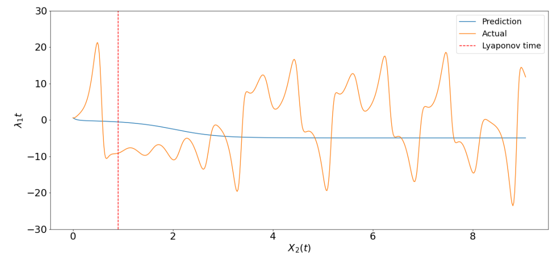
Table (1) represents the calculated Lyapunov exponents for the different settings. One can see that when the spectral radius of the reservoir is much smaller than unity, the exponent is negative but it decays very fast, therefor the reservoir will not be capable of reconstructing the long time correlations in data although the dynamics is stable. For spectral raduis close to unity, Lyapunov exponent has a negative value which does not decay for a long time. This means the dynamics is stable and can predict for more than one Lyapunov time scale depending on the structure of the reservoir. On the other hand in large spectral radius the exponent has a positive value which shows the instability of the reservoir. Fig.(2) depicts the behavior of $\frac{1}{n}\log\Lambda_1(n)$ against n for the three different spectral radii and their corresponding predictions.



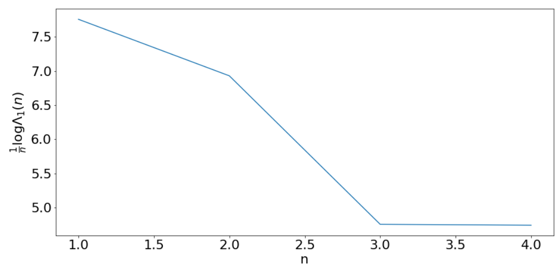
(a) Spectral radius = 1.2. Comparison of the resulting prediction with this setup is presented in Fig.(1). The exponent does not decay for a long time (at least as long as training time here that is checked) it remains negative and finite.



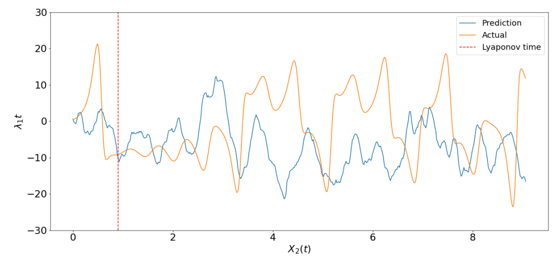
(b) Spectral radius = 0.01



(c) Spectral radius = 0.01



(d) Spectral radius = 1000



(e) Spectral radius = 1000

Figure 2: Behavior of Lyapunov exponent with different spectral radii.

Source code

```python
#!/usr/bin/env python3
# -*- coding: utf-8 -*-
"""
Created on Wed Apr 14 14:35:39 2021

@author: navid
"""


import numpy as np
import matplotlib.pyplot as plt
from sklearn.linear_model import Ridge


import matplotlib as mpl

mpl.rcParams['figure.titlesize'] = 25
mpl.rcParams['lines.linewidth'] = 1.5
mpl.rcParams['axes.labelsize'] = 22
mpl.rcParams['xtick.labelsize'] = 22
mpl.rcParams['ytick.labelsize'] = 22
mpl.rcParams['legend.fontsize'] = 22




class RESERVOIR():

    def __init__(self, data , n_reservoir=200, n_inputs=1 , n_outputs=1 ,
                 spectral_radius = 0.95, sparsity=0 , learning_rate=0.2 , n_trainings = 100):
        """
        Args:
            data: input time series consisting training and testing sets
            n_reservoir: number of reservoir neurons
            spectral_radius: spectral radius of the recurrent weight matrix
            sparsity: proportion of recurrent weights set to zero
            learning_rate: the learning rate to train using back-propagation
            n_trainings: number of trainings for the back propagion training
        """
        self.n_reservoir = n_reservoir
        self.spectral_radius = spectral_radius
        self.sparsity = sparsity
        self.n_inputs = n_inputs
        self.n_outputs = n_outputs
        self.data = data
        self.test_data = data[int(0.8*len(data)):]
        self.learning_rate = learning_rate
        self.n_trainings = n_trainings

    def initweights(self):
        # initialize reservoir weights:
        W = np.random.rand(self.n_reservoir, self.n_reservoir)*2-1
        W = W / (np.linalg.norm(W)+0.001)
        # delete the fraction of connections given by (self.sparsity):
        for i in range(self.n_reservoir):
            for j in range(self.n_reservoir):
                if np.random.uniform() > self.sparsity:
                    W[i,j] = 0
        # compute the spectral radius of these weights:
        radius = np.max(np.abs(np.linalg.eigvals(W)))
        # rescale them to reach the requested spectral radius:
        self.W = W * (self.spectral_radius / radius)

        # random input weights:
        self.W_in  = np.random.rand(self.n_reservoir , self.n_inputs )*0.2 - 0.1
        for i in range(self.n_reservoir):
            if np.random.uniform() < 0:
                self.W_in[i] = 0

```

```
        self.W_in = self.W_in/(np.linalg.norm(self.W_in)+0.001)
        # random output weights:
        self.W_out = np.random.rand(self.n_reservoir , self.n_outputs)*0.2 - 0.1
        self.W_out = self.W_out/(np.linalg.norm(self.W_out)+0.001)
        # initializing the neurons with ranodm values
        self.state = np.zeros((self.n_reservoir , 1))
        self.R = self.state
        self.J = 1
        self.landas = []

    def _update_weight(self, target_pattern, output_pattern):
        """Updates the output weights with gradient descent"""
        self.W_out += self.learning_rate*(target_pattern - output_pattern)*(self.state)

    def _predict(self, input_pattern):
        """given an input predicts the outpu using the trained weight"""
        b = np.dot(self.W, self.state) + np.dot(self.W_in, input_pattern)
        self.state = np.tanh(b)
        self.D = np.diag(1-np.tanh(b[:,0])**2)
        return np.dot(np.transpose(self.W_out),self.state)

    def _train(self, ridge=True):
        if ridge:
            self.model = Ridge(alpha=0.1)
            self.R = np.zeros((2000,self.n_reservoir))
            self.traj = [self.data[0]]
            for i in range(2000):
                self.R[i] = self.state.reshape(1,self.n_reservoir)
                self.traj.append(self._predict(self.data[i]))
                #self._single_val()
                #if i %100 == 0:
                #    print(i)

            self.model.fit(self.R, self.data[:2000])
            self.W_out = (self.model.coef_.reshape(self.n_reservoir , 1))
            self._test()
            self._plot(True)

        else:
            for train in range(self.n_trainings):
                self.R = np.zeros((2000,self.n_reservoir))
                self.traj = [self.data[0]]
                for i in range(2000):
                    self.R[i] = self.state.reshape(1,self.n_reservoir)
                    self.traj.append(self._predict(self.data[i]))
                    self._update_weight(self.data[i+1], self.traj[-1])
                if train%1==0:
                    print(f"train: {train} , learning_rate: {self.learning_rate}")
                    self._test()
                    self._plot(False)
                    self._plot_train(train)


    def _test(self):
        x = self.test_data[0]
        self.prediction = [x]
        for i in range(int(0.2*len(self.data))-1):
            self.prediction.append(self._predict(self.prediction[-1]))

    def _single_val(self):
        self.J = np.dot(np.dot(self.D , self.W) , self.J)
        self.landas.append(np.linalg.svd(self.J)[1][0])


    def _plot(self, saving):
        landa1 = 0.906
        T = np.linspace(0,10,500)*landa1
        plt.figure(figsize=(25,7))
        plt.plot(T , self.prediction , label='Prediction')
        plt.plot(T , self.test_data , label='Actual')
        plt.plot(np.ones(15)*landa1 , np.linspace(-30,30,15) , 'r--' , label='Lyaponov time')
        plt.legend(loc=4)
        plt.ylim(-30,30)
        plt.xlabel(r'$\lambda_1 t$')
```

```
        plt.ylabel(r'$y(t)$')
        if saving:
            plt.savefig('comparison.png')
        else:
            plt.show()

    def _plot_train(self, train):
        print(f'training {train} is doen!')
        plt.figure(figsize=(20,9))
        plt.plot(self.traj, label='train')
        plt.plot(self.data[:2000] , label='goal')
        plt.legend()
        plt.show()

    def _plot_lyapunov(self):
        self.landas = np.array(self.landas)
        plt.figure(figsize=(15,7))
        n = np.arange(1,len(self.landas)+1)
        plt.plot(n , 1/n*np.log(self.landas))
        plt.xlabel('n')
        plt.ylabel(r'$\frac{1}{n}\log \Lambda_1 (n)$')
        plt.savefig('Lyapunov.png')


    def _save(self):
        with open('W_out.npy' , 'wb') as f:
            np.save(f, self.W_out)
        with open('W_in.npy' , 'wb') as f:
            np.save(f, self.W_in)
        with open('W.npy' , 'wb') as f:
            np.save(f, self.W)
        with open('state.npy' , 'wb') as f:
            np.save(f, self.state)


    def run(self):
        for repeat in range(66,67):
            print(f'random seed : {repeat}')
            np.random.seed(repeat)
            self.initweights()
            self._train()
            #self._test()
            #print(f'repeat : {repeat}')
            #self._plot(True)
            #self._save()
            #self._plot_lyapunov()
```