Final Report

The effect of reservoir size on performance in Reservoir computing

Jiahao Cai
Supervisor: Maben Rabi

June 24, 2018

hh.se

Abstract

Reservoir computing is a framework for computation that may be viewed as an extension of Neural networks, we have explored the relationship between the reservoir size and the performance of the reservoir. More specifically, we apply the different reservoirs on different problems, e.g. sine functions, non-linear system such as Lorenz system and Rössler system, and Mountain car problem, which is a problem in the field of reinforcement learning. We use the reservoir computer to mimic a target system, and we evaluate the performance using root mean square (RMS) error. Through these experiments, we successfully find some trending to tune the parameters, e.g. reservoir size, average degree and leakage rate. We also discuss the limitations of the Reservoir computing.

Contents

1 Introduction

Recurrent neural networks (RNN) are efficient black-box models of nonlinear systems. Their feedback connections amongst themselves somehow consist a network which can predict time series, mimic a target system and so on. However, the available RNNs are not widely used in industry because of their slow convergence(Szita, Gyenes, and Lőrincz 2006).

Reservoir computing (Lukoševičius and Jaeger 2009) is a framework for computation that may be viewed as an extension of Neural networks, which considers much easier to train, because only the linear readout layer of the reservoir will be trained with teacher signals.

The reservoir computer has three components (Figure 1, Sande and Soriano 2017):

- an input layer of the feed-forward Neural network type,
- a reservoir layer of the recurrent Neural network type, and,
- a readout layer that is a weighted summer.

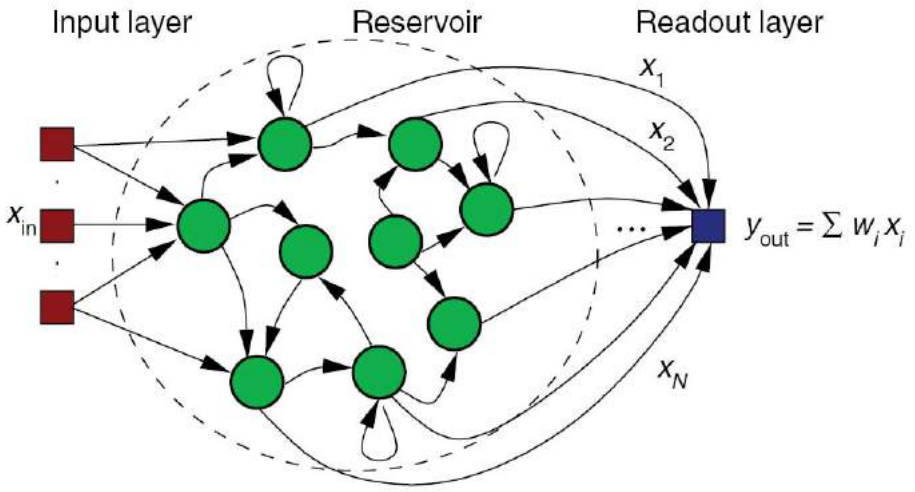


Figure 1: Components of Reservoir computing

The reservoir layer has three crucial properties:

1. it has a large number of nodes, and yet has sparse interconnections between them,
2. the nodes form a recurrent Neural network, because of feedback connections amongst themselves,
3. the node weight are updated during the learning phase using recursive and computationally efficient schemes.

The generalizing power of such learning machines is a subject of active research.

1.1 Problem formulation

Reservoir computing out-performs normal RNNs because of its randomly fixed middle layer: the reservoir layer. The random connectivity in the hidden layer does not give a clear insight into what is going on in the reservoir. Since the relationship between the size of the reservoir and the threshold of Reservoir computing is still veiled, researches using Reservoir computing tend to use sparse hidden layer for better performance. However, in a practical scenario, it is always better to use an appropriate reservoir to solve a certain problem without wasting computing resources. Also we need to explore how fast the reservoir should evolve according to the complexity of different problems, and whether we should use some popular techniques in normal RNNs such as learning rate decay.

More specifically, in order to create a minimum reservoir according to the complexity of different problems, the following research questions need to be solved:

1. How will N influence the performance of Reservoir computing? Does a larger N always bring better performance? Does there exist a relationship between N and the performance of the reservoir?
2. How will the D influence the performance of Reservoir computing? Does there exist a relationship between the D and the performance?
3. What is the appropriate N when facing problems with different computational complexity? How can one describe the complexity of a learning problem?
4. What is the appropriate α when facing problems with different evolve speed? How can one describe the evolve speed of a learning problem?

1.2 Objectives

This project aims to explore the relationship between the size of the hidden layer and the power of Reservoir computing using different computational complexity examples. First, we will start with simple functions, mainly includes sine function and its derivations; then we will try to simulate Lorenz system and Rössler system with the reservoir; we will also test reservoir with mountain car problem.

This project has its own limitation, because it will explore the relationship using more actual performance tests instead of mathematical deduction.

1.3 Related work

Previous work in Reservoir computing seeks to find a way to tune the reservoir, its applicable area and limitation. Since Reservoir computing is introduced with its methods, architectures and tuning principles (Lukoševičius and Jaeger 2009), this research area has been prolific. The trends of Reservoir computing has been discussed (Lukoševičius, Jaeger, and Schrauwen 2012), a tutorial on the general application and implementation of Reservoir computing has been given (Schrauwen, Verstraeten, and Van Campenhout 2007), and there is also a toolbox for Reservoir computing available (Verstraeten et al.

2012). Towards specific problems, Lu et al. 2017 applies Reservoir computing to Rössler system, Lorenz system and Kuramoto-Sivashinsky equations.

The prior work mostly uses a typically sparse large reservoir, our job is trying to find the relationship between the size of reservoir and its performance when applying to a certain problem.

There has also been work in applying Reservoir computing in the field of Reinforcement learning (RL), which focus on maintaining a long term history in the RNN and making the internal representation has Markov property at the same time (Bakker 2002 and Szita, Gyenes, and Lőrincz 2006). Our experiment towards RL is trying to learn a solution to mountain car problem (Moore 1990) generated by Q-learning (Watkins and Dayan 1992).

2 Reservoir computing

Typically, a reservoir computer has three components: a linear input layer with M input nodes, a recurrent reservoir network with N nodes, and a linear output layer with P output nodes. In this environment, the reservoir dynamics is defined as:

$$\boldsymbol{r}(t + \Delta t) = (1 - \alpha)\boldsymbol{r}(t) + \alpha \; tanh(\boldsymbol{A}\boldsymbol{r}(t) + \boldsymbol{W}_{in}\boldsymbol{u}(t) + \xi 1)$$

where

- $\boldsymbol{r}(t)$ is a N-dimensional internal state vector of the reservoir,
- α is the leakage rate, it controls the speed of how fast the reservoir evolves, higher value means faster speed,
- $\mathbf{A}$ is an adjacency matrix, which is built from a sparse random Erdős-Rényi matrix, the value of non-zero element are randomly drawn independently from a uniform distribution between -1 and 1,
- $\boldsymbol{W}_{in}$ is a linear input weight matrix, which makes the M-dimensional input the suitable size to be fed in the reservoir, the non-zero elements are randomly chosen from a uniform distribution between -0.5 and 0.5,
- $\boldsymbol{u}(t)$ is the input signal, which is M-dimensional,
- $\xi 1$ is a bias term where 1 denotes a vector of ones and ξ is a scalar constant.

In addition to the parameters above, there are some other parameters which will influence the power of reservoir when constructing one. For example,

- Ω is architecture, which is the way of how the internal nodes connected in the reservoir,
- D is average degree, $D * N$ equals to the number of edges among nodes in the reservoir,
- β is a typically small number, which is the parameter of ridge regression, it represents the extend of penalizing reservoir to generalize.

When doing experiments towards different problems, these parameters will be fixed separately. For example, for all experiments with sine function and Lorenz system, α is fixed as 0.3 and 1.0 separately because sine function is considered evolving relatively slower than Lorenz system.

The output layer is a P-dimensional vector, which is calculated by the following equation:

$$\hat{\boldsymbol{s}}(t) = \boldsymbol{W}_{out}\boldsymbol{r}(t)$$

where $\hat{\boldsymbol{s}}(t)$ is the approximated value of the input signal, $\boldsymbol{W}_{out}$ stands for the output weight, which is calculated using ridge regression after training and $\boldsymbol{r}(t)$ is the internal state vector at time t.

The training has two phases, initial period ($t1$) and training period ($t2$). $t1$ is comparably short, we simply feed the input signal into the reservoir; in $t2$, we feed the subsequent input signal into the reservoir and collect the $\boldsymbol{r}(t)$ (internal state vector of reservoir) as $\boldsymbol{R}$ and $\boldsymbol{s}(t)$ (output of reservoir) as $\boldsymbol{S}$.

Ridge regression will be used for training $\boldsymbol{W}_{out}$:

$$\boldsymbol{W}_{out} = \boldsymbol{S}\boldsymbol{R}^T(\boldsymbol{R}\boldsymbol{R}^T + \beta\boldsymbol{I})^{-1}$$

where $\boldsymbol{I}$ is the $N * N$ identity matrix.

There will also be a period called test period ($t3$), we shall use root mean square (RMS) error to evaluate the performance of a reservoir in $t3$. We will collect the output of reservoir and compare with the teacher signal, lower RMS error means better performance. In order to test the performance of mimicking a target system, such as sine functions and Lorenz system, we will feed a subsequent signal as an activation, and we feed the output of the reservoir as the input of the reservoir, so that the reservoir does not need external source anymore. We keep collecting output of the reservoir until $t3$ finishes, then we will compare with the teacher signal and get the RMS error. On the other hand, if the reservoir is used for predicting, we shall keep feeding the subsequent input signal and collect the output until $t3$ finishes, then calculate the RMS error.

3 Experiments

3.1 Sine function and its derivations

All the experiments in this section empirically use the following parameters:

- number of nodes: $N = [2, 150]$,
- average degree: $D = \sqrt{N}$,
- leakage rate: $\alpha = 0.3$,
- regression parameter: $\beta = 1 \times 10^{-8}$,

- architecture: Ω is a fixed structure,
- initial period: $t1 = 1000$,
- training period: $t2 = 5000$,
- test period: $t3 = 500$.

3.1.1 $\boldsymbol{u}(t) = \sin(t)$

In this set-up, when $N < 4$, the reservoir cannot simulate $\sin(t)$ at all (Figure 2) while $N = 4$ the reservoir can give a good simulation (Figure 3).

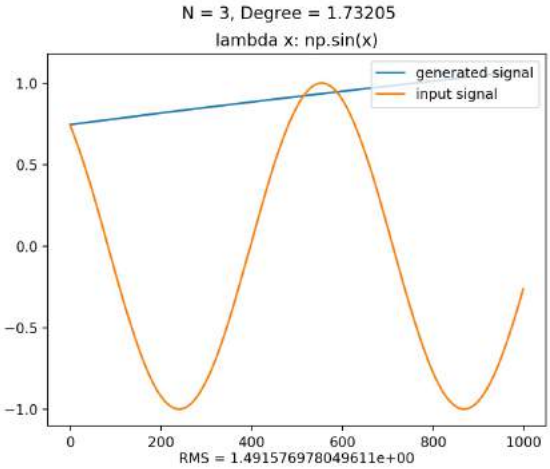


Figure 2: $N = 3$, simulating $\sin(t)$

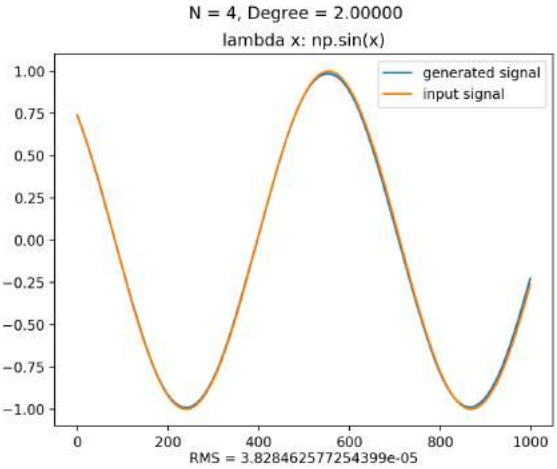


Figure 3: $N = 4$, simulating $\sin(t)$

In addition to making $D = \sqrt{N}$, we have also tested $D = 0.1$, $D = 0.5$, $D = 2$, $D = \log(N)$ and $D = N$ with different N (Figure 4). We found that when $D = 0.1$, the RMS error is almost invariable, it barely changes as the almost-flat red line shown in the Figure; when $D = 0.5$, it give some good RMS errors, but most of the blue line is still overlapped with the red line; when $D = 2$, $D = \log N$, $D = \sqrt{N}$ and $D = N$, as the N is increasing, the reservoir does not show the ability to better simulate $\sin(t)$, on the contrary, it gives higher overall RMS error when increasing N, and the RMS errors of these different D are surprisingly similar, they overlap too much so that we can only see the orange line in the Figure.

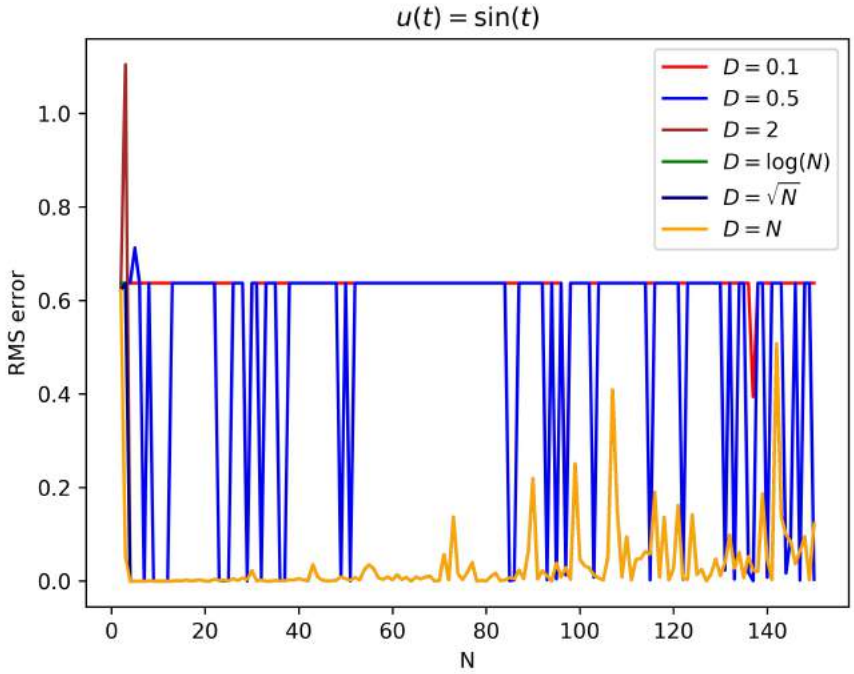


Figure 4: RMS error of reservoir with 2-150 nodes trying to simulate $\sin(t)$

3.1.2 $u(t) = \sin(5t) + 10$

$\sin(5t) + 10$ has shorter period and larger value comparing to $\sin(t)$. In this set-up, when $N = 4$, the reservoir is not enough to give a good simulation (Figure 5) while $N = 5$ is enough (Figure 6).

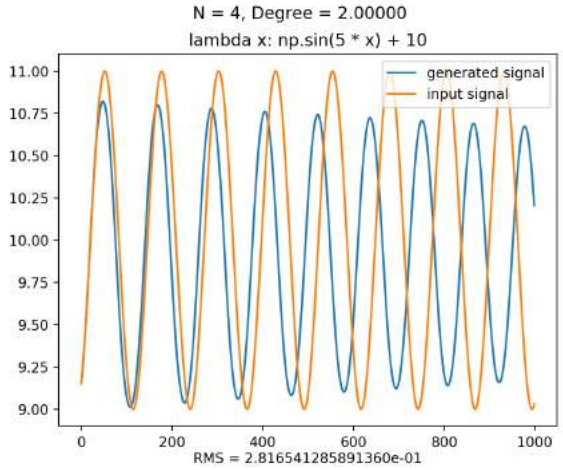


Figure 5: $N = 4$, simulating $\sin(5t) + 10$

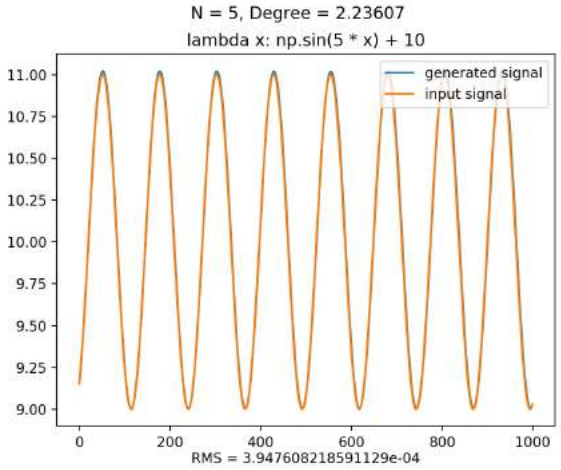


Figure 6: $N = 5$, simulating $\sin(5t) + 10$

Similar to the $\sin(t)$ example before, we have calculated the RMS error with different N and D (Figure 7). We found that as N increases, the overall RMS error is also increasing, which is similar to what happened before.

When $D = 0.1$, the RMS error is just a almost-flat red line in the Figure; when $D = 0.5$, the RMS error becomes unstable, it can give both good simulation and bad simulation, most of the blue line is still overlapped with the red line; when $D = 2$, $D = \log N$, $D = \sqrt{N}$ and $D = N$, the reservoir gives very good RMS error when N is small, but when N begins to increase, the overall RMS error is increasing as well, and the RMS errors of these different D are also very similar, they overlap too much so that we can only see the orange line in the Figure.

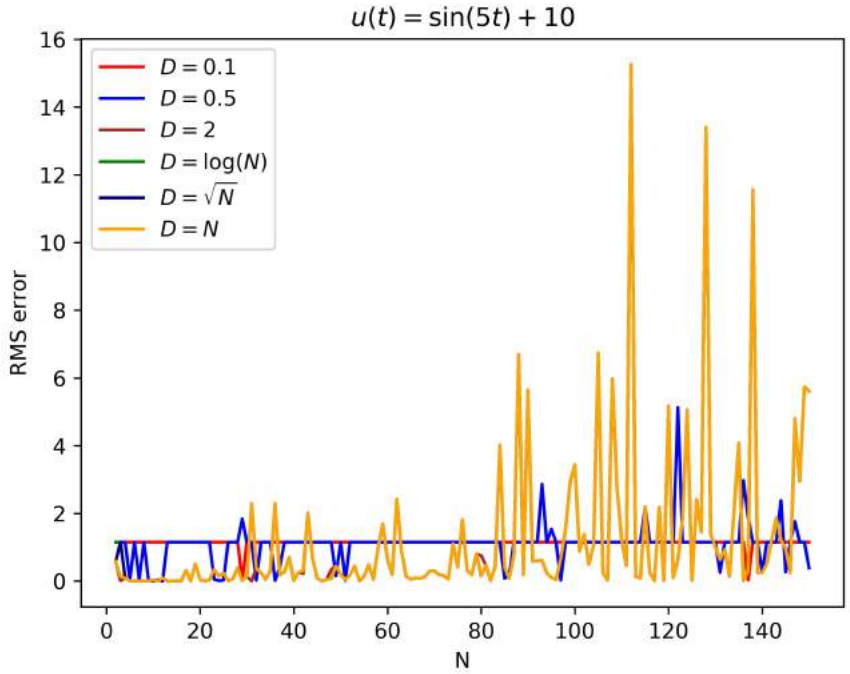


Figure 7: RMS error of reservoir with 2-150 nodes trying to simulate $\sin(5t) + 10$

3.1.3 $\boldsymbol{u}(t) = \sin(t) + \sin(\pi * t)$

There are no available ways found yet to simulate $\sin(t) + \sin(\pi * t)$, $N = 5$ or $N = 50$ gives similar results (Figure 8 and Figure 9), as a matter of fact, the RMS error is always high (above 1) when $N = [2, 150]$ (Figure 10).

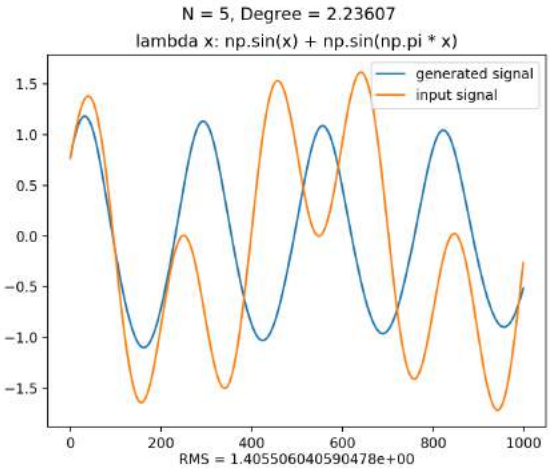


Figure 8: $N = 5$, simulating $\sin(t) + \sin(\pi * t)$

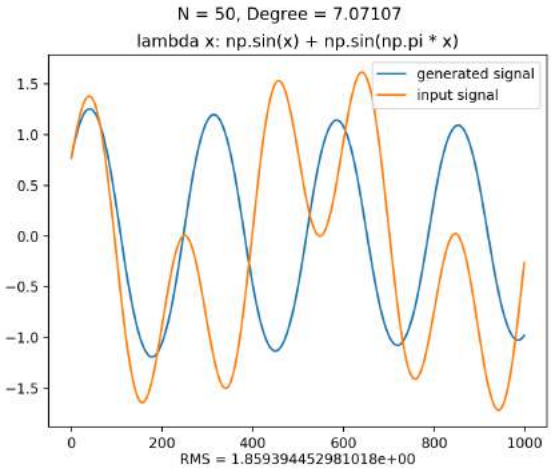


Figure 9: $N = 50$, simulating $\sin(t) + \sin(\pi * t)$

Similar to the examples before, we have calculated the RMS error with different N and D (Figure 10). We found that as N increases, the overall RMS error is not changing a lot, we find it is because that the reservoir have not learned this $\sin(x) + \sin(\pi * x)$ at all. Also, when $D = 0.1$, the RMS error is just a almost-flat red line in the Figure; when $D = 0.5$, it gives some good RMS errors, most of the blue line is still overlapped with the red line; when $D = 2$, $D = \log N$, $D = \sqrt{N}$ and $D = N$, the RMS errors of these different D are also very similar, they overlap too much so that we can only see the orange line in the Figure.

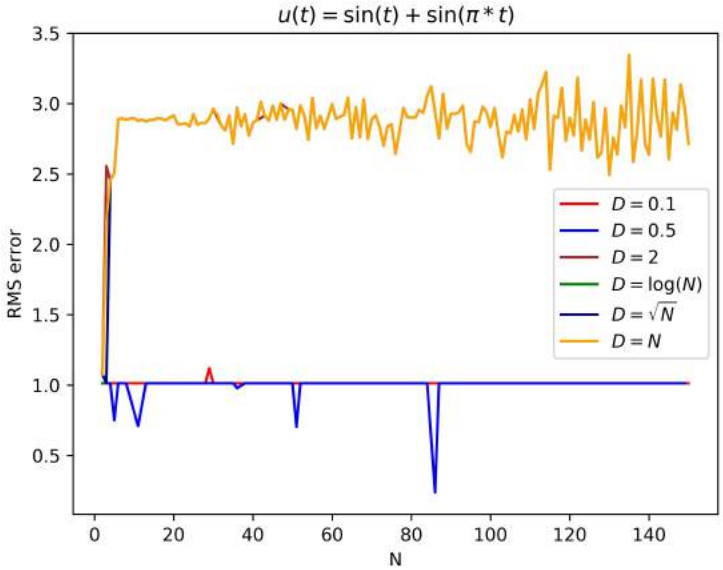


Figure 10: RMS error of reservoir with 2-150 nodes trying to simulate $\sin(t) + \sin(\pi * t)$

3.1.4 $\boldsymbol{u}(t) = \sin(\sin(t))$

$\sin(\sin(t))$ has the exact period length as $\sin(t)$ but different value. In this set-up. When $N = 3$, the reservoir is not good enough to give a good simulation (Figure 11) while $N = 4$ is enough (Figure 12).

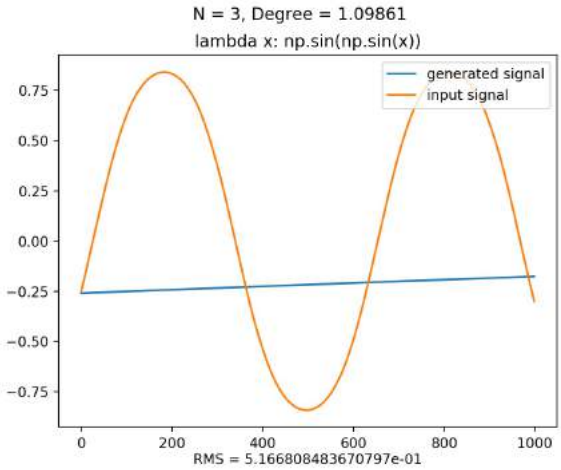


Figure 11: $N = 3$, simulating $\sin(\sin(t))$

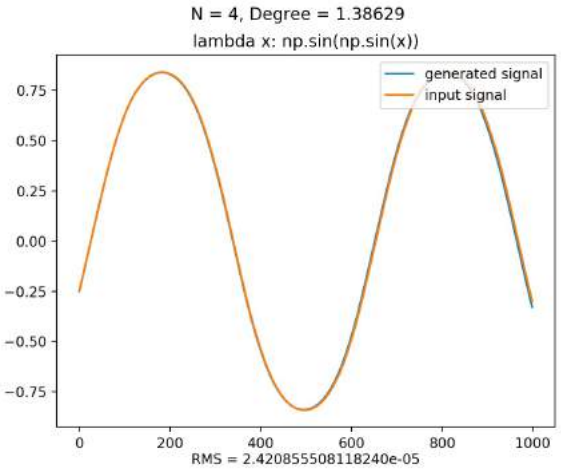


Figure 12: $N = 4$, simulating $\sin(\sin(t))$

However, while increasing N, the reservoir gives very high RMS error, e.g. $N = 23$ (2.799×10^{52}), $N = 27$ (2.766×10^{58}), $N = 36$ (5.900×10^{43}) and so on. We eliminated this problem by turning down α to 0.05 (Figure 13). Similar to the reservoirs before, when $D = 0.1$, the RMS error barely changes as the red line shown; when $D = 0.5$, it gives some good RMS errors, most of the blue line is still overlapped with the red line; when $D = 2$, $D = \log N$, $D = \sqrt{N}$ and $D = N$, the reservoir gives very good RMS error when N is small, but when N begins to increase, the overall RMS error is increasing as well, and the RMS errors of these different D are also very similar, they overlap too much so that we can only see the orange line in the Figure.

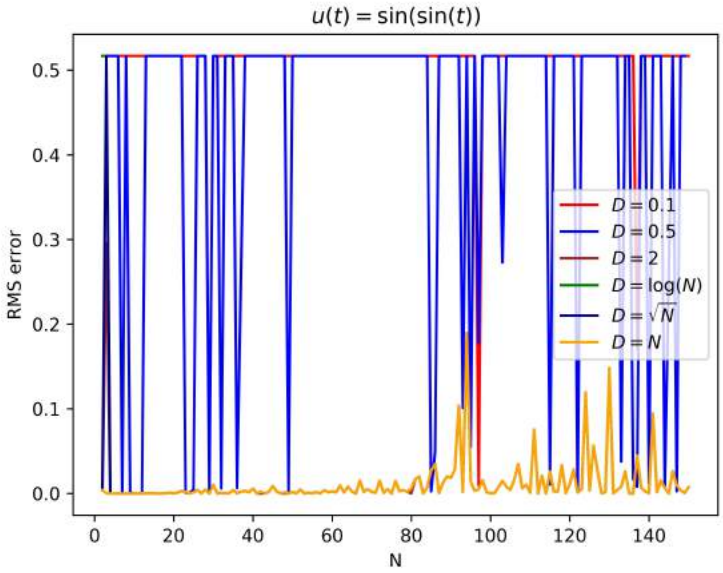


Figure 13: RMS error of reservoir with 2-150 nodes trying to simulate $\sin(\sin(t))$

3.2 Lorenz system

Lorenz system is a chaotic system which is described by the equations:

$$\frac{\mathrm{d}x}{\mathrm{d}t} = \sigma(y - x), \quad \frac{\mathrm{d}y}{\mathrm{d}t} = x(\rho - z) - y, \quad \frac{\mathrm{d}z}{\mathrm{d}t} = xy - \beta z,$$

where $\sigma = 10$, $\beta = 8/3$ and $\rho = 28$ in this set-up.

All the experiments in this section empirically use the following parameters:

- number of nodes: $N = [100, 500]$
- average degree: $D = \sqrt{N}$
- leakage rate: $\alpha = 1.0$,
- regression parameter: $\beta = 1 \times 10^{-8}$,
- architecture: Ω is a fixed structure,
- initial period: $t1 = 1000$,
- training period: $t2 = 5000$,
- test period: $t3 = 500$.

Lorenz system is considered far more complicated than sine functions, thus the reservoir needs more nodes. In this experiment, we feed the reservoir with (x, y, z) at the same time, and evaluate the performance using the RMS error of each signal. We have tested the performance of reservoir when $N = 100$ (Figure 14), $N = 200$ (Figure 15), $N = 300$ (Figure 16), $N = 400$ (Figure 17), $N = 500$ (Figure 18) and $N = 600$ (Figure 19).

As we can see in these figures, $N = 400$ is the most appropriate size, reservoir with too few nodes ($N = 100$, $N = 200$, $N = 300$) or too many nodes ($N = 500$, $N = 600$) cannot give a good simulation towards Lorenz system. Note that the input signal and the generated signal are either the same or the opposite, that is because the Lorenz system is invariant under the transformation $(x, y, z) \to (-x, -y, z)$.

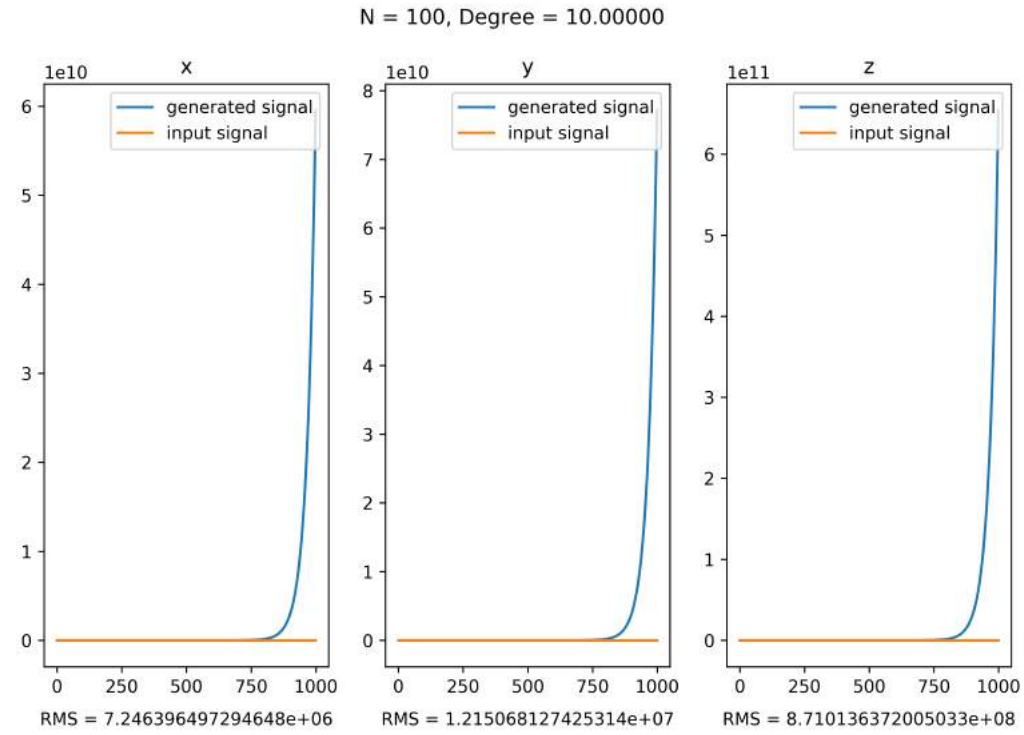


Figure 14: RMS error of reservoir with 100 nodes trying to simulate Lorenz system

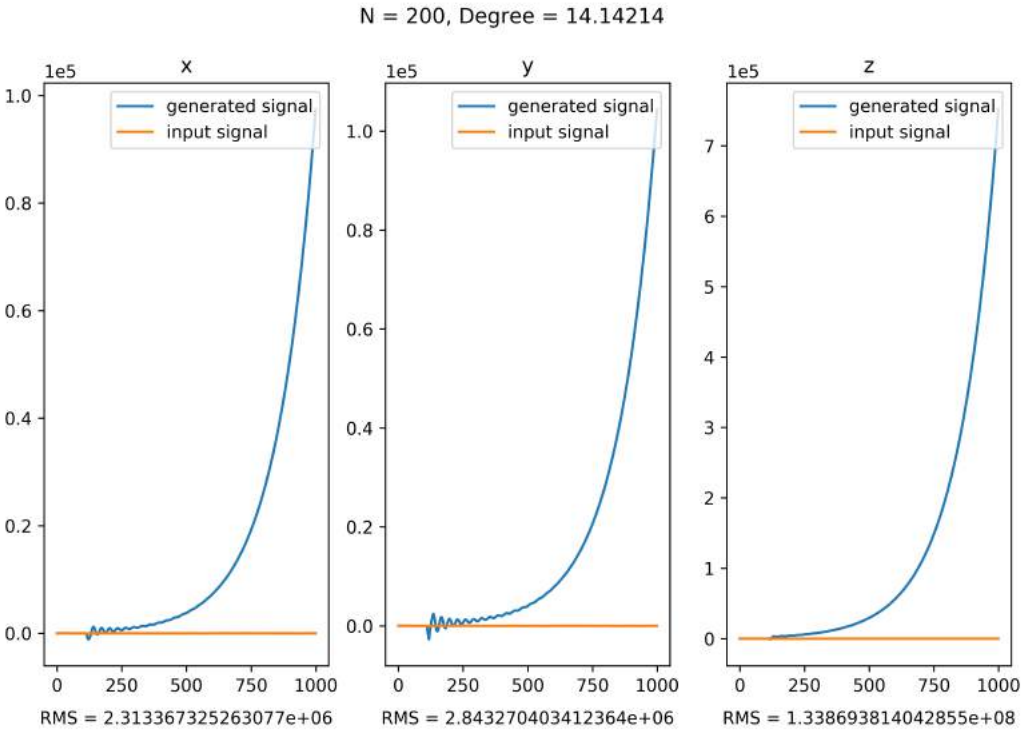


Figure 15: RMS error of reservoir with 200 nodes trying to simulate Lorenz system

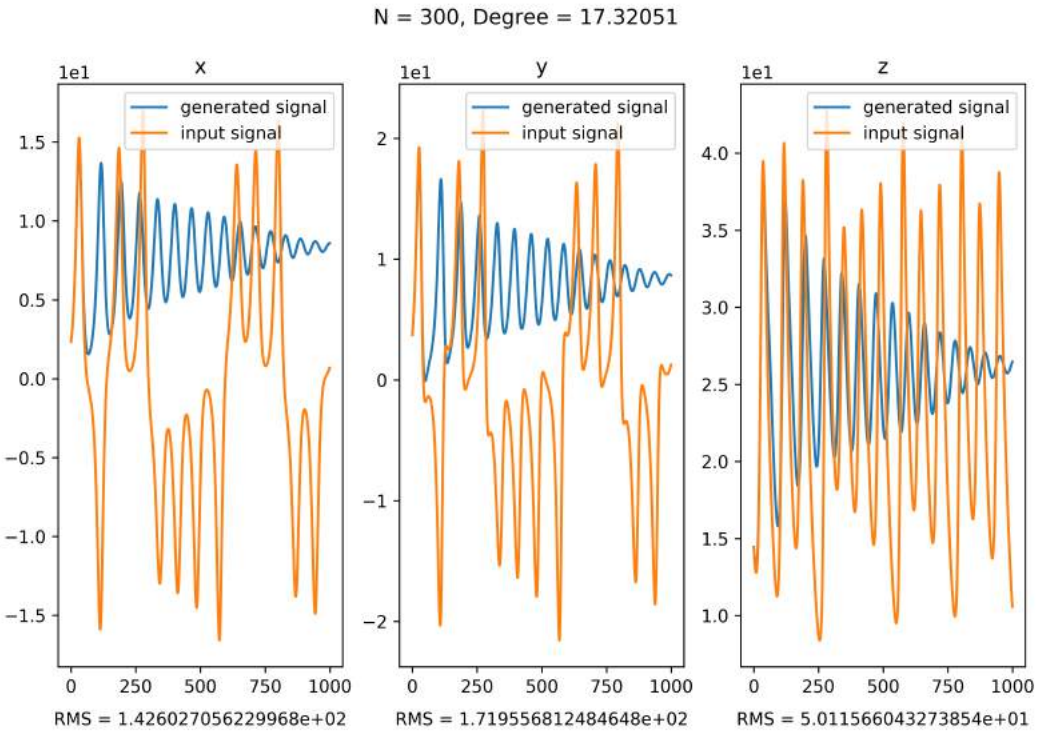


Figure 16: RMS error of reservoir with 300 nodes trying to simulate Lorenz system

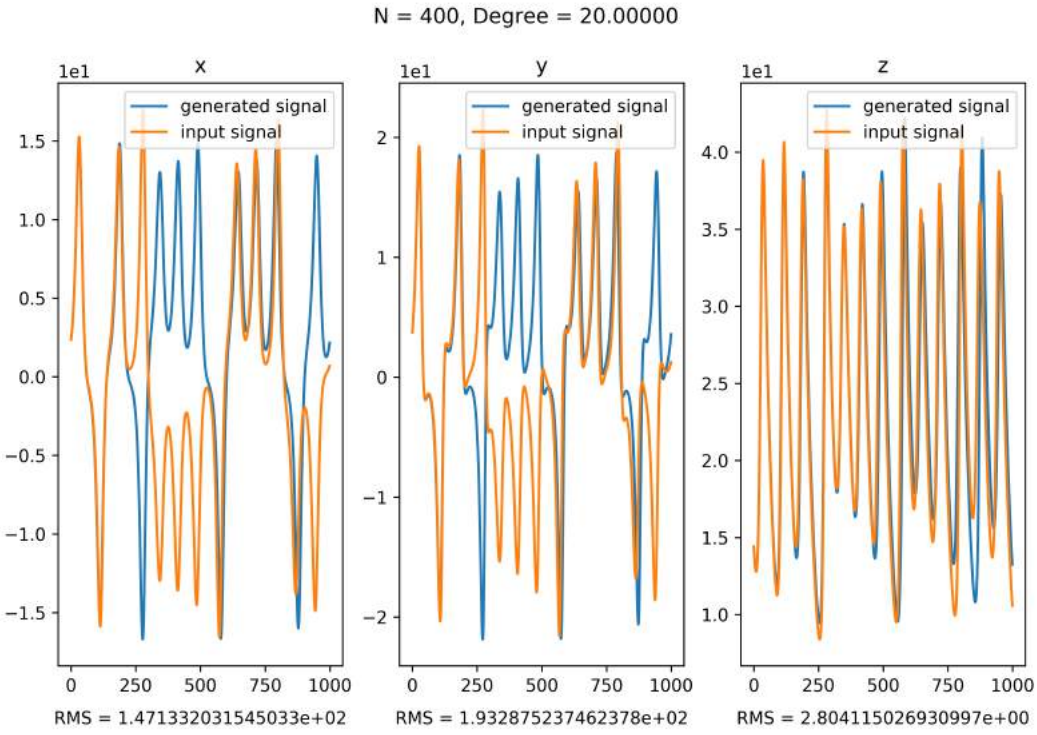


Figure 17: RMS error of reservoir with 400 nodes trying to simulate Lorenz system

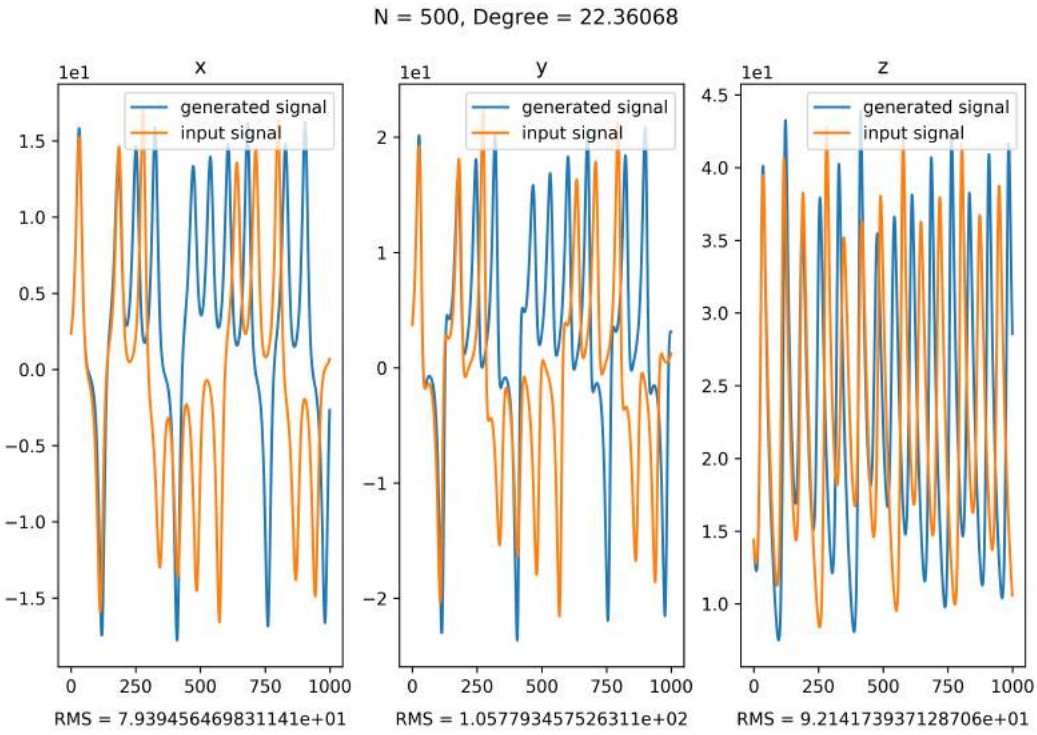


Figure 18: RMS error of reservoir with 500 nodes trying to simulate Lorenz system

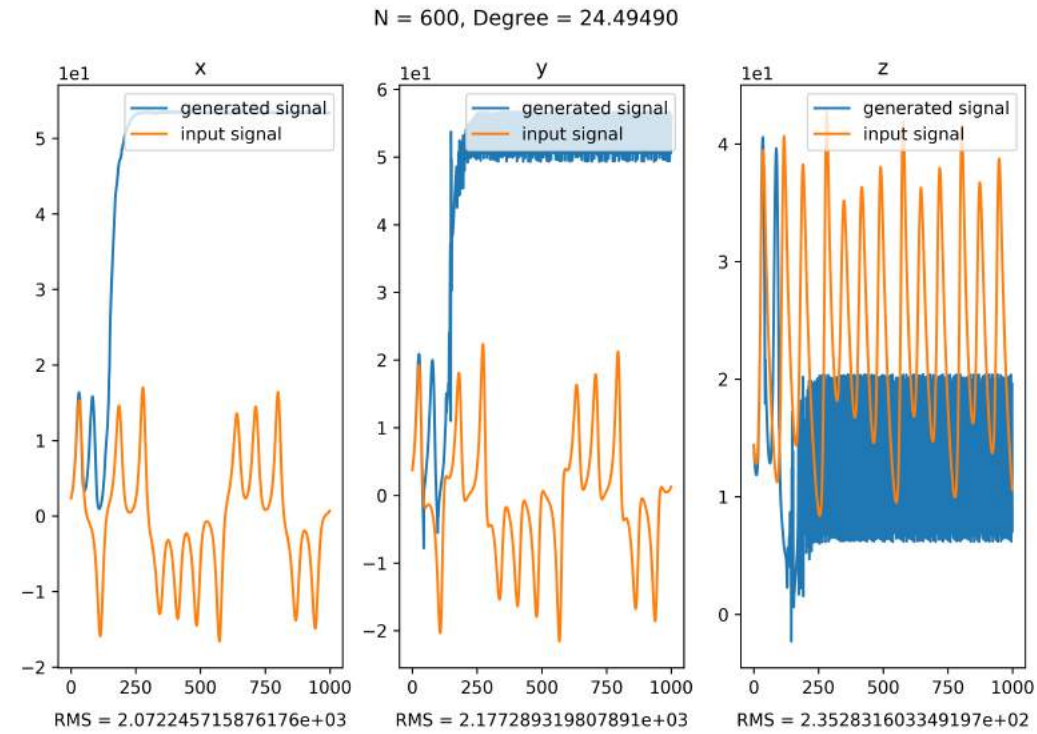


Figure 19: RMS error of reservoir with 600 nodes trying to simulate Lorenz system

3.3 Rössler system

Rössler system is also a chaotic system, which is described by the equations:

$$\frac{\mathrm{d}x}{\mathrm{d}t} = -y - z, \quad \frac{\mathrm{d}y}{\mathrm{d}t} = x + ay, \quad \frac{\mathrm{d}z}{\mathrm{d}t} = b + z(x - c),$$

where $a = 0.5$, $b = 2.0$ and $c = 4.0$ in this set-up.

The parameters in this experiment is the same as the parameters in the Lorenz system experiment except we set $t2 = 6000$ and $t3 = 3000$. In this experiment, we feed the reservoir with (x, y, z) at the same time, and evaluate the performance using the RMS error of each signal. We have tested the performance of reservoir when $N = 50$ (Figure 20), $N = 100$ (Figure 21), $N = 150$ (Figure 22), $N = 200$ (Figure 23), $N = 250$ (Figure 24) and $N = 300$ (Figure 25).

As we can see in these figures, $N = 100$ is the most appropriate size, reservoir with too few nodes ($N = 50$) or too many nodes ($N = 150$, $N = 200$, $N = 250$, $N = 300$) cannot give a good simulation towards Rössler system.

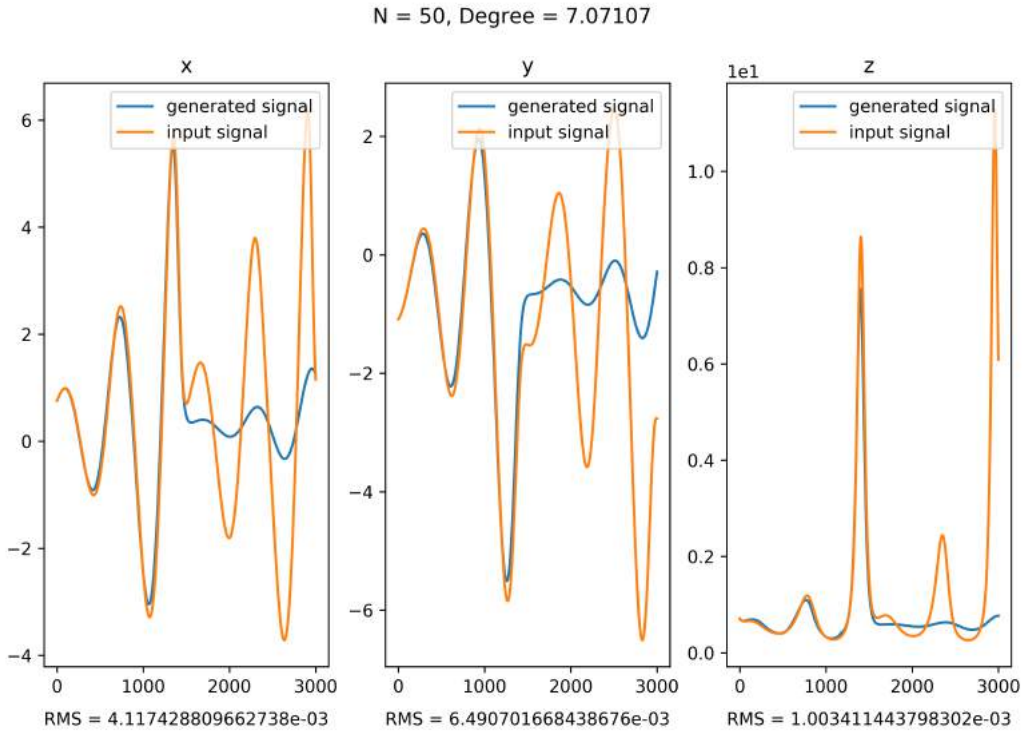


Figure 20: RMS error of reservoir with 50 nodes trying to simulate Rössler system

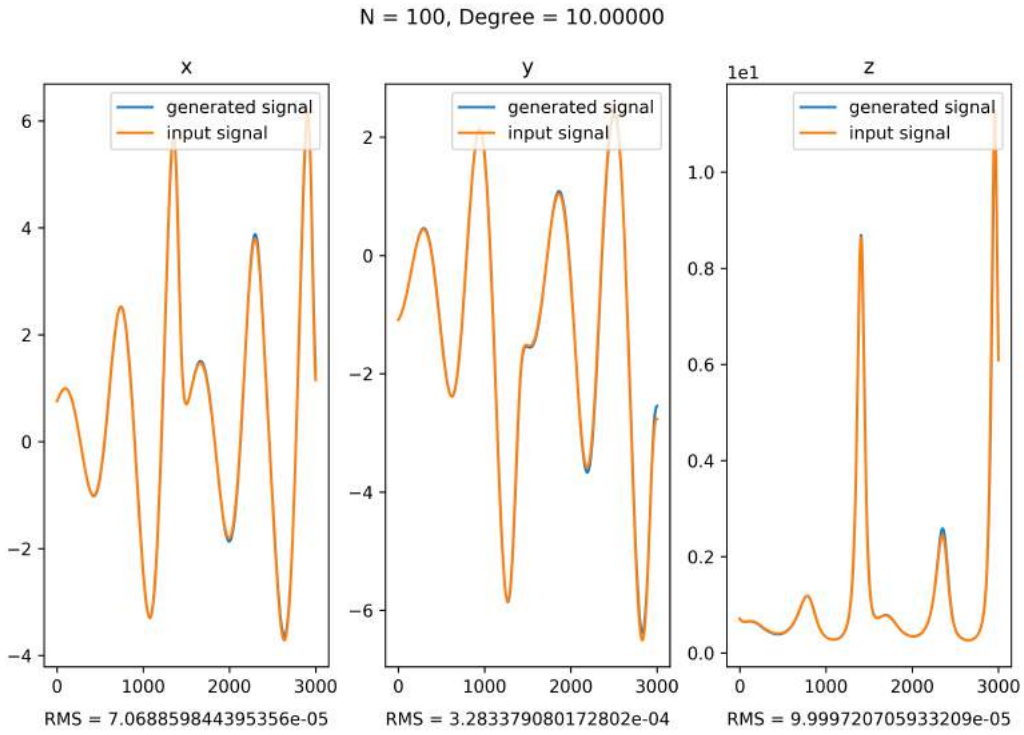


Figure 21: RMS error of reservoir with 100 nodes trying to simulate Rössler system

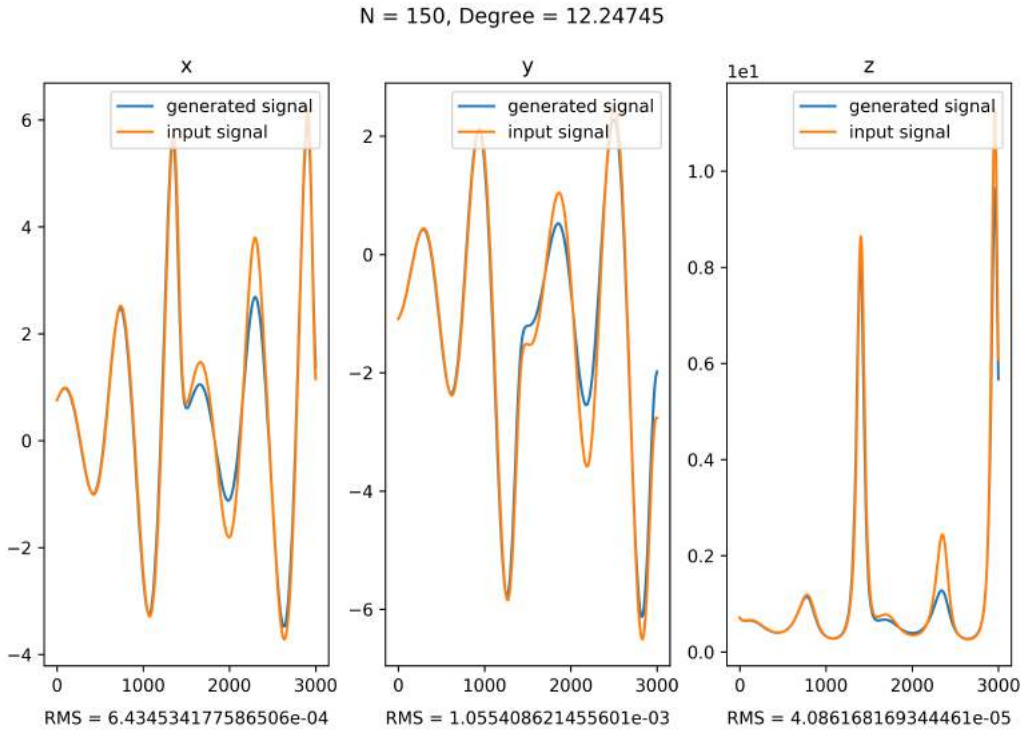


Figure 22: RMS error of reservoir with 150 nodes trying to simulate Rössler system

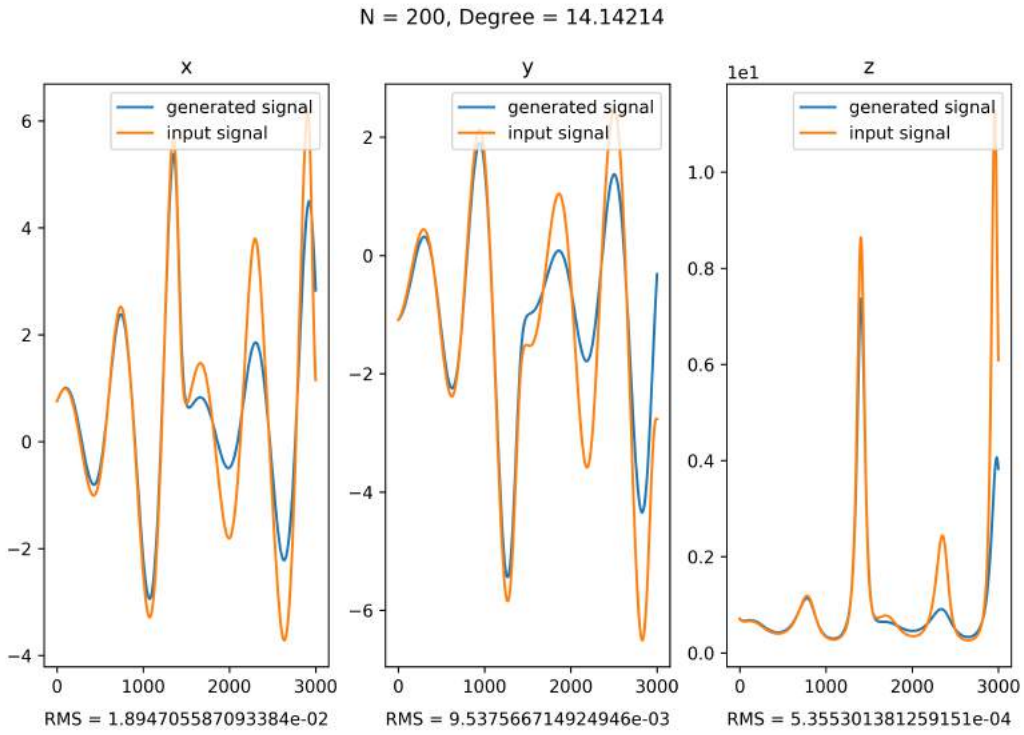


Figure 23: RMS error of reservoir with 200 nodes trying to simulate Rössler system

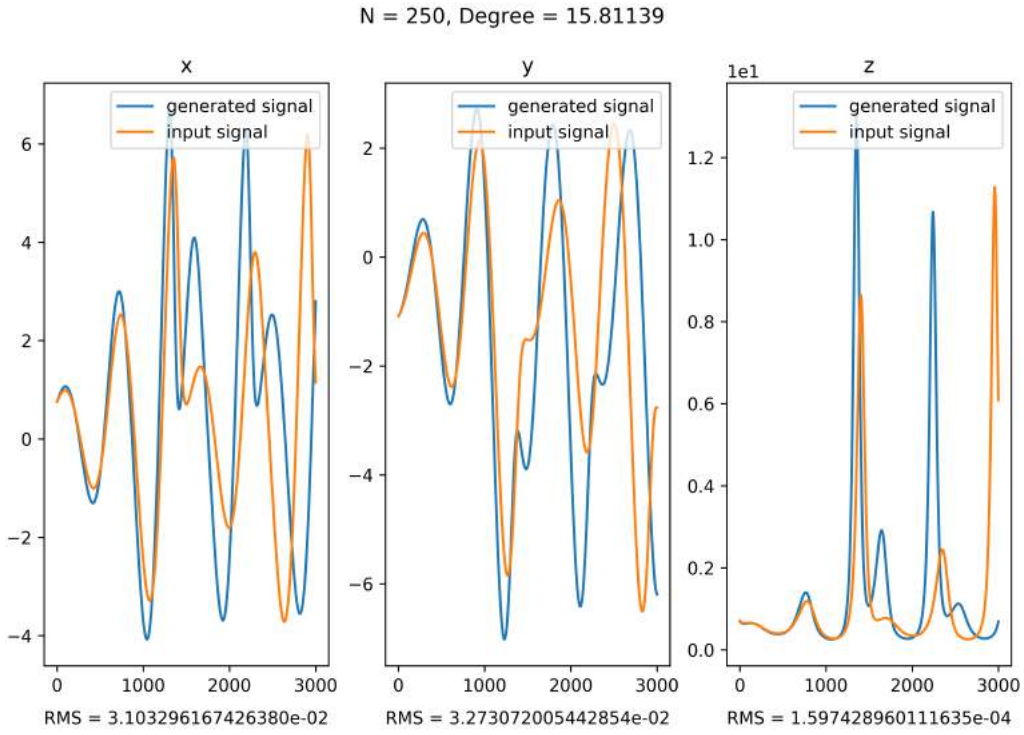


Figure 24: RMS error of reservoir with 250 nodes trying to simulate Rössler system

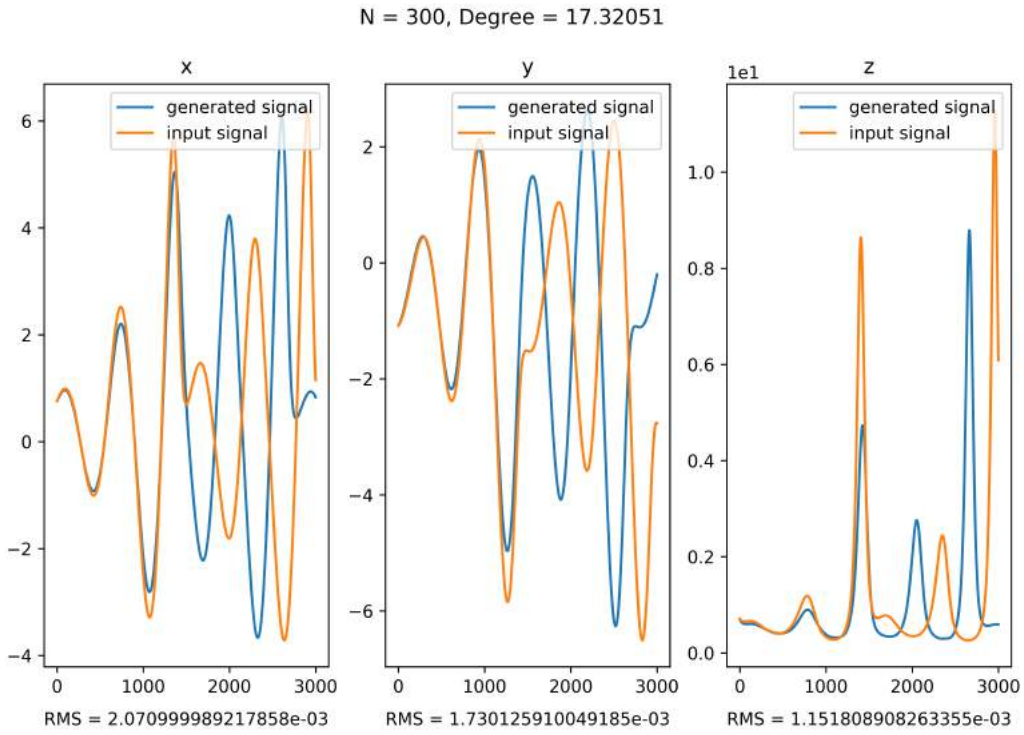


Figure 25: RMS error of reservoir with 300 nodes trying to simulate Rössler system

3.4 Mountain car problem

Mountain car problem (Figure 26, Moore 1990) is commonly applied in the field of reinforcement learning (RL), which requires the agent to learn how to drive the car up to the right hill. The environment in this experiments is provided by OpenAI Gym(Brockman et al. 2016). The agent has to take three kinds of actions based on different states, which is driving left, driving right and not using the engine at all. Note that the engine of car is not strong enough, so the car cannot simply accelerate up the right hill.

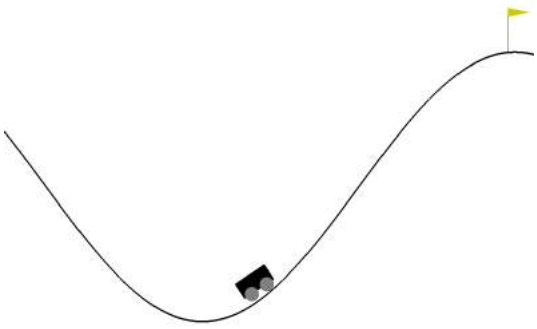


Figure 26: The agent will learn how to drive the car up to the right hill

The mountain car problem is defined as the following equations:

$$Velocity = Velocity + Action * 0.001 + \cos(3 * Position) * (-0.0025)$$
$$Position = Position + Velocity$$

where $Velocity = (-0.07, 0.07)$ and $Position = (-1.2, 0.6)$.

(1). Reservoir computing with Q-learning

Reservoir computing cannot apply to RL problems directly, because Reservoir computing does not have a way of evaluating the long-term reward, which is the job of value function in the RL.

A alternative way to solve the mountain car problem using Reservoir computing is solving through Q-learning (Watkins and Dayan 1992), which can maintain a Q-table as the value function, which will use the state (s) as the column and the action (a) as the row. In this case, the state of the car has two parts, velocity (v) and position (p), so the Q-table will be a three-dimensional table, represented as $Q(v, p, a)$, every entry will be initilized as 0. In the beginning, we select a random state s_t and take a random action a_t into the next state s_{t+1}, in this process, we will observe a reward r_t, r_t will be added to $Q(v_t, p_t, a_t)$.

The selection of next action will be following this algorithm:

$$Q(v_t, p_t, a_t) \leftarrow (1 - \alpha) * Q(v_t, p_t, a_t) + \alpha * (r_t + \gamma * max(Q(v_{t+1}, p_{t+1}, a))$$

where α is the learning rate and γ is the discount factor.

In the training period, we use the following parameters:

- action: 3 actions, 0 for drive left, 1 for stay still and 2 for drive right,
- state: since v (-0.07, 0.07) and p (-1.2, 0.6) are continuous, transformations are needed to map them as discrete. Here we map v and p as 40 states, so the size of $Q(v, p, a)$ is $(40 * 40 * 3)$,
- reward: every state will give the same reward -1, staying longer means more negative rewards, so the optimal strategy will be getting into the terminal state as soon as possible,
- episode: the total training episode is 100 000, the current episode is represented as i. In a single episode, it's possible that the agent cannot get to the terminal state forever, we provide maximum 200 trials in each episode, if the agent uses up 200 trials, we will skip to the next episode, so the minimum reward of each episode is -200,
- learning rate: we are using a techique called learning rate decay, which means to use a comparably large learning rate first, and keep turning it down while running the agent. $\alpha = 1.0$ at first, and is updated using as $\alpha = max(0.003, 1.0 * (0.85^{\lfloor i/100 \rfloor}))$,
- discount factor: $\gamma = 1.0$.

When the training finishes, each entry $Q(v, p, a)$ stores the accumulated reward for the action a under the state v and p, which stands for the quality of a, because less negative reward means better action under that state. Then we choose the optimal action under each state and finally each entry will contain the optimal action in that state.

(2). Result

With the Q-table, running the mountain car problem for 1000 times gives average reward of -121.25. We feed the velocity and position of the car into the reservoir, record the output, and train the output layer with the values in the Q-table.

All the experiments in this section empirically use the following parameters:

- number of nodes: $N = [5, 100]$,
- average degree: $D = \sqrt{N}$,
- regression parameter: $\beta = 1 \times 10^{-8}$,
- architecture: Ω is a fixed structure,
- initial period: $t1 = 5$ episodes,
- training period: $t2 = 500$ episodes,
- test period: $t3 = 100$ episodes.

As the result shows (Figure 27), just like the sine functions before, the performance does not increase as the size of reservoir increases. The reservoir with 15 nodes gives the best reward when $\alpha = 0.1$, -171.41, however, it is still very far away from the reward generated by the Q-table (-121.25). When $\alpha = 0.1$ and $\alpha = 0.2$, they show better ability to learn when N is small and worse ability to learn when N is large comparing to $\alpha = 0.01$ and $\alpha = 0.04$. When $\alpha = 0.01$ and $\alpha = 0.04$, the average reward is confined to a certain range, $\alpha = 0.04$ shows better overall learning ability comparing to $\alpha = 0.01$.

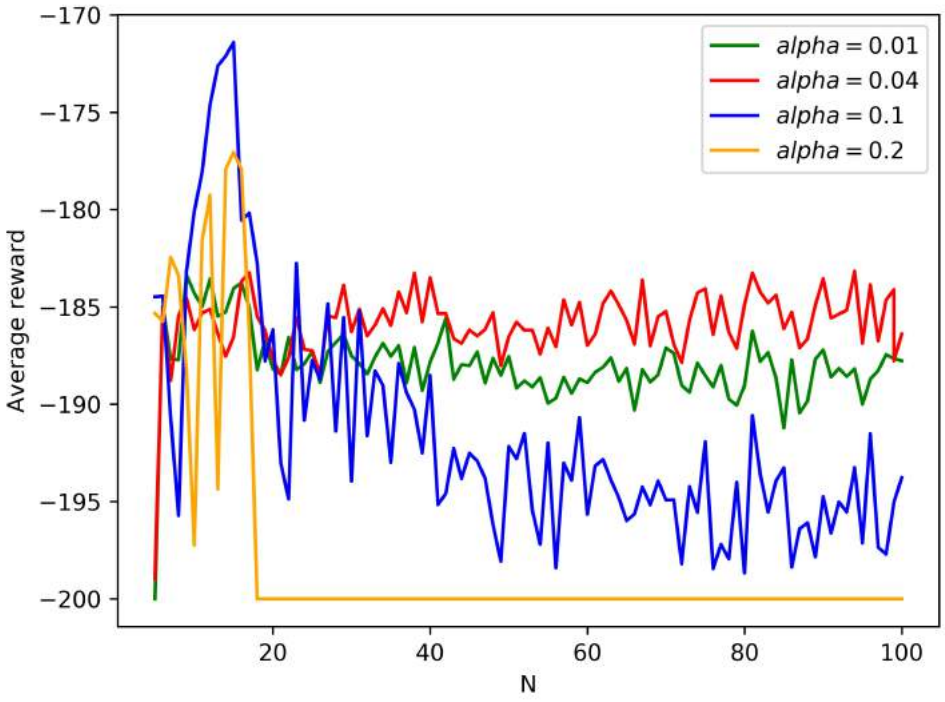


Figure 27: reward of reservoir with 5-100 nodes trying to simulate mountain car

4 Discussion

In this work, we tested the performance of reservoir with different size when applying to three different problems. During this process, we found something worth further exploring.

4.1 Complexity with size of reservoir N

According to the experiment with sine functions, we find that it is not always better to use a larger reservoir. In fact, if N exceeds a certain number, the overall RMS error will increase as N increases. Take $\boldsymbol{u}(t) = \sin(5t) + 10$ as an example, similar RMS errors are given by reservoirs with N,

- 8 nodes: 9.349×10^{-4},
- 43 nodes: 4.933×10^{-4},
- 90 nodes: 9.405×10^{-4}.

These nodes above give quite good RMS error, if we can find the relationship between the complexity of problem and N, we can solve problems more easily.

However, a certain complexity should not be mapped to many reservoirs with very different N. If we calculate the average RMS error together with the adjacent nodes, the result will be

- 5-11 nodes: 6.608×10^{-3},
- 40-46 nodes: 6.540×10^{-2},
- 87-93 nodes: 9.871×10^{-1}.

In this case, there exists a range of nodes which gives much better performance than others, we think we should find such a range of nodes towards a certain complexity instead of several very different N.

Similar situation happens in $\boldsymbol{u}(t) = \sin(t)$, $\boldsymbol{u}(t) = \sin\sin(t)$ and mountain car problem as well.

4.2 Performance with average degree D

In the experiments with sine functions, we found that when $D = 0.1$ and $D = 0.5$, the result is different according to the problems. For example, in $\boldsymbol{u}(t) = \sin(t)$ and $\boldsymbol{u}(t) = \sin\sin(t)$, when $D = 0.1$ and $D = 0.5$, the RMS error is higher than the RMS error when D is larger; in $\boldsymbol{u}(t) = \sin(t) + \sin(\pi * t)$, when $D = 0.1$ and $D = 0.5$, the RMS error is lower than the RMS error when D is larger; in $\boldsymbol{u}(t) = \sin(5t) + 10$, when $D = 0.1$ and $D = 0.5$, the RMS error is very close to the RMS error when D is larger.

Also, we can find that when D exceeds a certain value, D is not affecting the performance of the reservoir any more, it is possible that the reservoir is always trying to activate the same nodes and same edges when applying to the same problem regardless of other nodes

and edges. So if we can figure out the minimum value of D we should use when applying to a certain problem, the reservoir will be running much faster, because a sparse hidden layer is always takes much less computational resource than a fully connected hidden layer.

4.3 Evolve speed with leakage rate α

Leakage rate α plays an important role in Reservoir computing because it controls the speed of how fast a reservoir evolves. In the experiments, we empirically use $\alpha = 1.0$ on Lorenz system, $\alpha = 0.3$ on sine functions and $\alpha = 0.04$ on mountain car problem, because we find that Lorenz system evolves faster than sine functions and sine functions evolve faster than mountain car problem.

The following simple experiment may demonstrate the evolve speed intuitively. When we use a 6-node reservoir with $\alpha = 0.6$ on $\boldsymbol{u}(t) = \sin(t)$ and $\boldsymbol{u}(t) = \sin(t/4)$, the result shows that the reservoir can give a good simulation to $\boldsymbol{u}(t) = \sin(t)$ but not $\boldsymbol{u}(t) = \sin(t/4)$ (Figure 28 & 29).

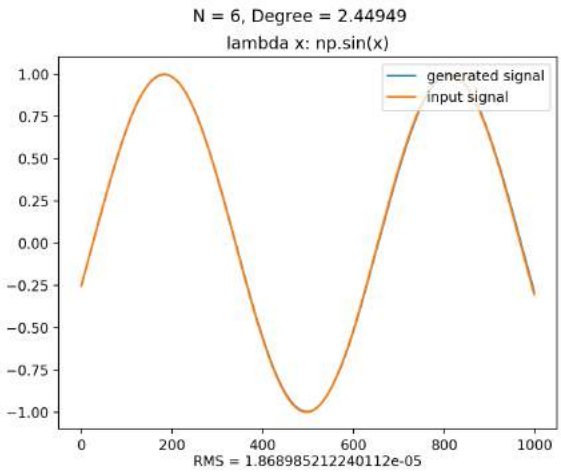


Figure 28: $\sin(t)$ with leakage rate 0.6

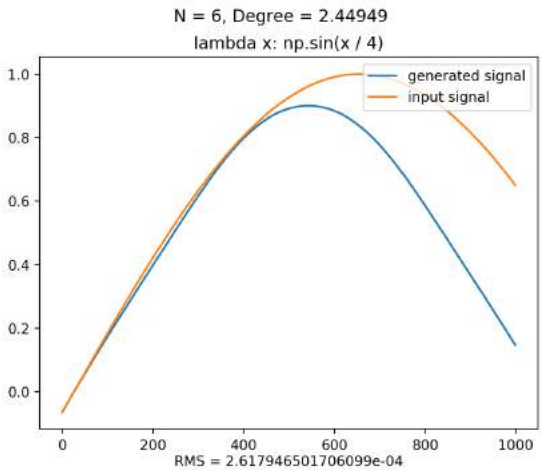


Figure 29: $\sin(t/4)$ with leakage rate 0.6

However, if we change the α to 0.29, the reservoir will give a good simulation to $\boldsymbol{u}(t) = \sin(t/4)$ (Figure 30).

We find that $\sin(t/4)$ evolves slower than $\sin(t)$ as the Figure 31 shows, thus the reservoir needs smaller α to learn $\sin(t/4)$. If we can find a way to description the evolve speed of a certain problem and find its relationship with the α, this will help us solve problems with Reservoir computing.

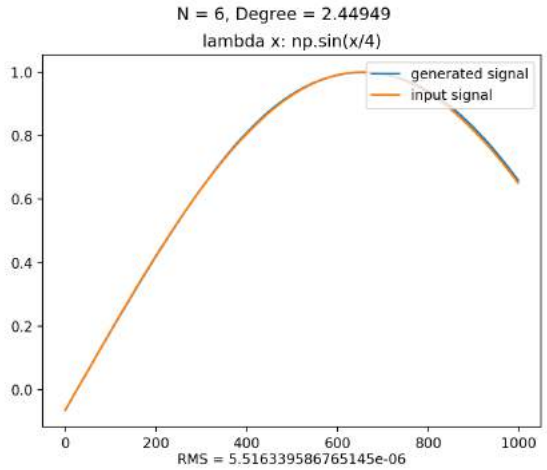


Figure 30: $\sin(t/4)$ with leakage rate 0.29

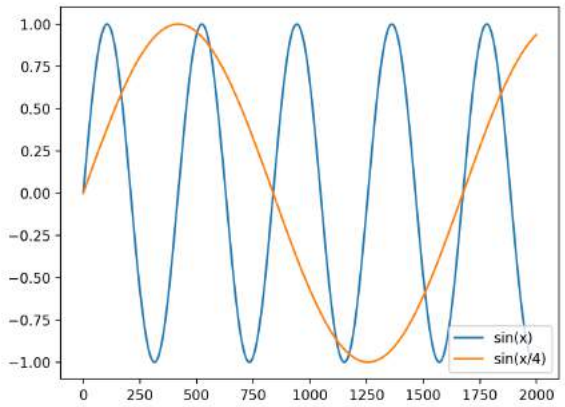


Figure 31: $\sin(t)$ and $\sin(t/4)$

4.4 Why $\boldsymbol{u}(t) = \sin(t) + \sin(\pi * t)$ is hard to learn

An intuitive explanation is that $\boldsymbol{u}(t) = \sin(t) + \sin(\pi * t)$ is a non-periodical function, and the reservoir is very hard to learn a non-periodical function (except for very simple function like $\boldsymbol{u}(t) = t$). In the common cases, we feed the Neural network with many training examples, for instance, we feed the Neural network with many period of sine function and each period can be viewed as a training example. However, if the function is non-period, we can assume that we are feeding an infinite training example into the Neural network, and it cannot find a repeatable pattern, as a result, the Neural network cannot learn this function.

Similar situation happens to non-periodical functions such as when $\boldsymbol{u}(t) = \sin(t) + \cos(\pi * t)$ and $\boldsymbol{u}(t) = t * \sin(t)$ as well.

4.5 Future work

In this project, we find some trending of how to tune a reservoir, e.g. tune the leakage rate towards problems with different speed of evolving, but it is based on limited experiments and it is lack of solid math proof. As a result, more experiments are needed and we need to see it in a more mathematical way. For example, we need to test more different functions.

Besides, all the experiments in this project are based on a fixed reservoir architecture, same regression parameter, and same way to calculate the average degree, it is possible that any change to these parameters can lead to an totally different conclusion. We will try to explore these parameters in the future as well.

References

Szita, István, Viktor Gyenes, and András Lőrincz (2006). "Reinforcement learning with echo state networks". In: *International Conference on Artificial Neural Networks*. Springer, pp. 830–839.

Lukoševičius, Mantas and Herbert Jaeger (2009). "Reservoir computing approaches to recurrent neural network training". In: *Computer Science Review* 3.3, pp. 127–149.

Sande Guy, Daniel Brunner Van der and Miguel C. Soriano (2017). "Advances in photonic reservoir computing". In: *Nanophotonics*.

Lukoševičius, Mantas, Herbert Jaeger, and Benjamin Schrauwen (2012). "Reservoir computing trends". In: *KI-Künstliche Intelligenz* 26.4, pp. 365–371.

Schrauwen, Benjamin, David Verstraeten, and Jan Van Campenhout (2007). "An overview of reservoir computing: theory, applications and implementations". In: *Proceedings of the 15th European Symposium on Artificial Neural Networks. p. 471-482 2007*, pp. 471–482.

Verstraeten, David et al. (2012). "Oger: modular learning architectures for large-scale sequential processing". In: *Journal of Machine Learning Research* 13.Oct, pp. 2995–2998.

Lu, Zhixin et al. (2017). "Reservoir observers: Model-free inference of unmeasured variables in chaotic systems". In: *Chaos: An Interdisciplinary Journal of Nonlinear Science* 27.4, p. 041102.

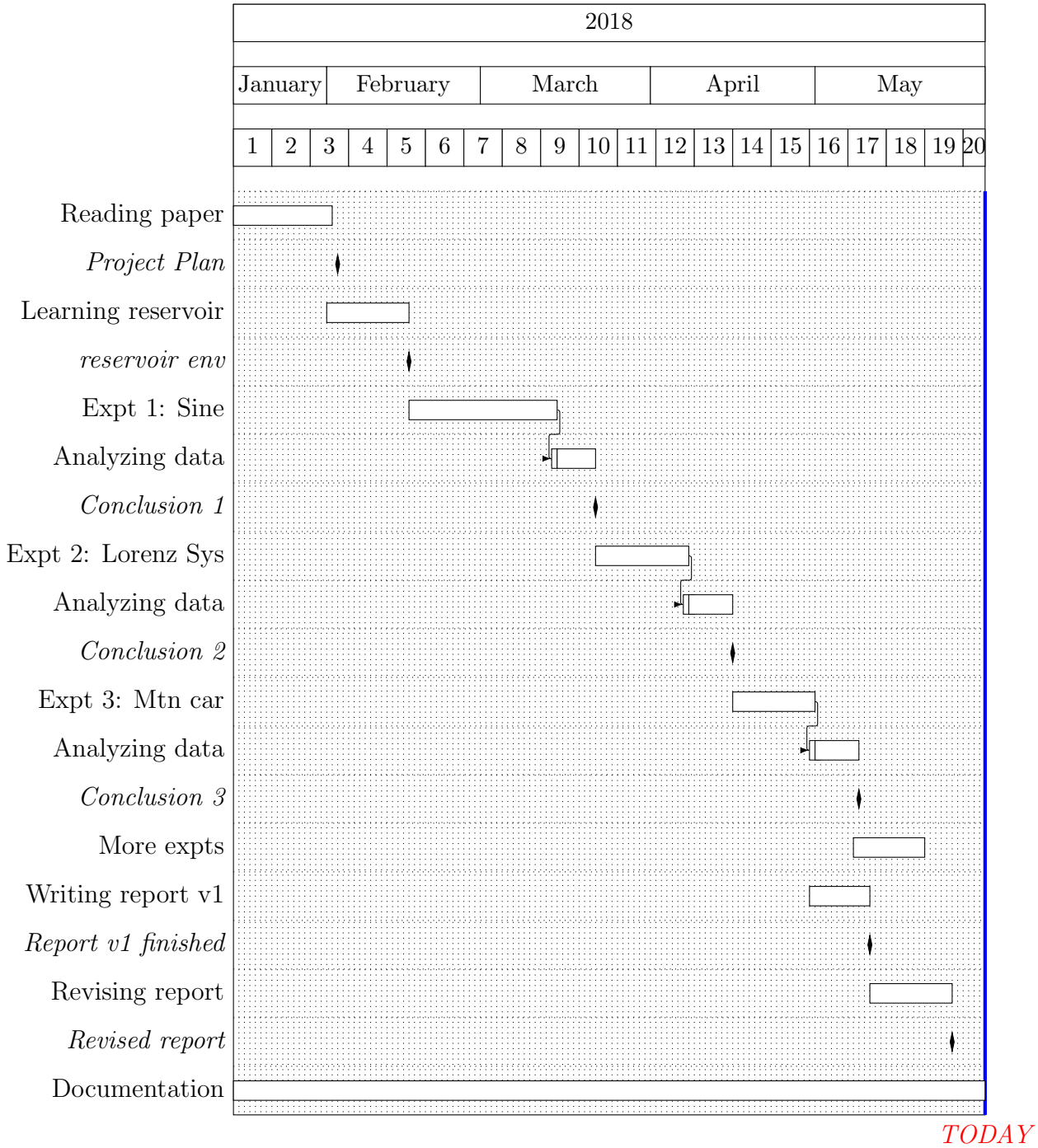
Bakker, Bram (2002). "Reinforcement learning with long short-term memory". In: *Advances in neural information processing systems*, pp. 1475–1482.

Moore, Andrew William (1990). *Efficient Memory-based Learning for Robot Control*. Tech. rep.

Watkins, Christopher JCH and Peter Dayan (1992). "Q-learning". In: *Machine learning* 8.3-4, pp. 279–292.

Brockman, Greg et al. (2016). "OpenAI Gym.(2016). arXiv". In: *arXiv preprint arXiv:1606.01540*.

5 Time-plan


2018
January
February
March
April
May
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20
Reading paper
Project Plan
Learning reservoir
reservoir env
Expt 1: Sine
Analyzing data
Conclusion 1
Expt 2: Lorenz Sys
Analyzing data
Conclusion 2
Expt 3: Mtn car
Analyzing data
Conclusion 3
More expts
Writing report v1
Report v1 finished
Revising report
Revised report
Documentation
TODAY