Quantum Dot Reservoir Computing

Project Notes

August 30, 2024

Contents

1 Abstract

Information processing and analysis of time series are crucial in many applications but often face constraints such as high computational complexity. Quantum reservoir computing, which combines a reservoir of quantum hardware with a simple artificial neural network, offers a promising solution. The main idea is to utilize the complicated dynamics generated by quantum systems to create reservoirs that handle more challenging temporal learning tasks.

In this project, we simulate a system of interacting coupled quantum dots connected to electronic leads as a quantum reservoir. We use a Lindblad master equation approach to calculate the transient dynamics and transport through the systems and evaluate its performance through several benchmark tests.

2 Background

Modelling of time series and temporal learning tasks, are important in may applications, all from modelling of biological systems to financial economic forecasting. A neural network model traditionally used for temporal learning tasks and time series prediction is Recurrent Neural Networks (RNNs), see figure 1A for an example. The feedback connections in RNNs allow the network output to be dependent on past inputs, making them well-suited for modeling of time series data. However, RNNs often encounter computational constraints during training. When training the network, calculating gradients over multiple time steps is required, leading to high computational complexity and problems with exploding and vanishing gradients.

Various techniques have been developed to improve RNN's, for example Long Short-Term Memory (LSTM) networks. Another approach is Reservoir computing, where only the output layer of the network is trained, while the rest remains untrained. The untrained part is often referred to as the reservoir. An example of a reservoir computing model is illustrated in figure 1B.

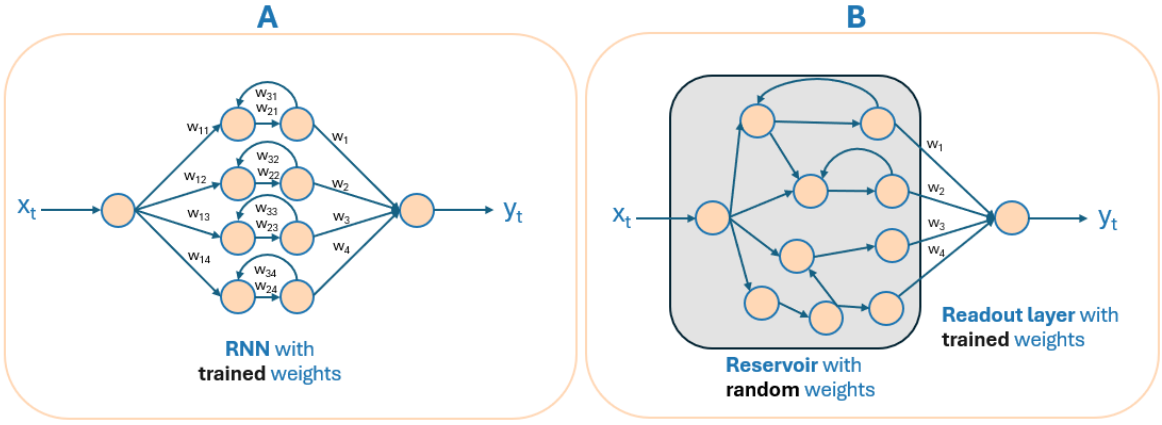

Figure 1: Figure A illustrates an example of a RNN. Figure B illustrates a reservoir computing model, where only the outputl layer is trained.

Instead of using an artificial neural network as a reservoir, physical dynamical systems can serve as reservoirs. In this case, we can utilize the natural dynamics of the physical system to store and encode data. To do this, input is fed into the physical reservoir, and the output is obtained by measuring the reservoir's state. The output is then passed through a single-layer network. This model is illustrated in figure 2.

There are multiple advantages with using a reservoir computing model instead of an RNN. First off, only the output layer needs to be trained, which is computationally easy. A second benefit is that if we have different targets we want to train for the same input. We only need to feed the input to the reservoir and measure the reservoir's output once. Then the reservoir's output can be used to train different output layers for different targets. This is commonly referred to as multitasking. More information on the benefits of physical reservoir computing can be found in for example [1] and [2].

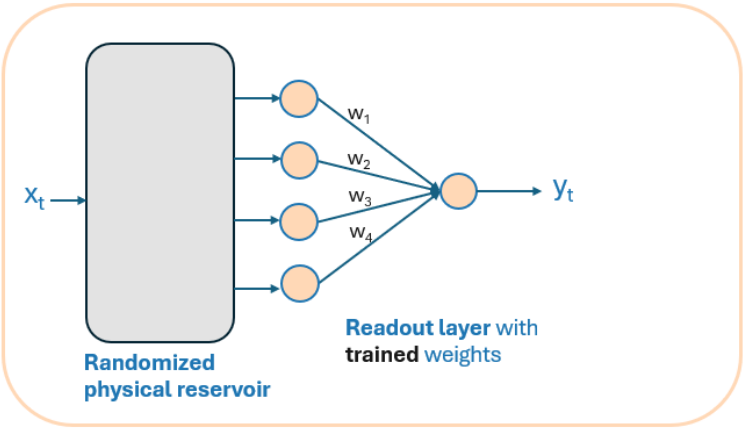

Figure 2: Input is sent to the reservoir and the reservoir's state is measured as output. The reservoir's the output is then sent to a single layer neural network giving a final output.

In our case, we use a reservoir consisting of four coupled quantum dots connected to electronic leads, as illustrated in figure 3. Input is fed to the reservoir by modifying the chemical potential of the leads according to the input function, which is scaled by an input weight for each lead. The reservoir's output is the current in each lead at a number of time steps. The current is then processed by a trained linear layer to produce the final output.

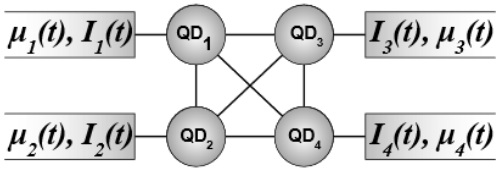

Figure 3: Figure of the reservoir design.

3 Reservoir Dynamics

The reservoir is consisting of four spinless quantum dots connected to electronic leads, as shown in figure 3. The system of quantum dots is described by the Hamiltonian:

$$H_{dots} = \sum_i \epsilon_i n_i + \sum_{i \neq j} t_{ij} d_i^\dagger d_j + \sum_{i \neq j} U_{ij} n_i n_j \tag{1}$$

where the energy levels (ϵ_i), tunneling terms (t_{ij}) and Coulomb interaction terms (U_{ij}) are randomized from a uniform distribution.

Reservoir dynamics are simulated by solving the Lindblad master equation (equations 2 and 3). A parameter "evolution rate" α is added in equation 2 to re-scale the time of the quantum dot dynamics. By increasing α, the dynamics will be "faster", meaning that the system decays to the steady state solution faster.

$$\dot{\rho} = \alpha \mathcal{L}(t) \rho \tag{2}$$

$$\mathcal{L}(t)\rho = -\frac{i}{\hbar}[\tilde{H}, \rho] + \sum_i \gamma_i(t) \left(L_i(t) \rho L_i^\dagger(t) - \frac{1}{2} \left\{ L_i^\dagger(t) L_i(t), \rho \right\} \right) \tag{3}$$

The chemical potential of the leads is changed as a function of time according to an input function. Using QMEQ, an open-source Python package [3], the system Lindbladian ($\mathcal{L}(t)$) is calculated as a function of time. Next, the time-dependent density matrix is obtained by numerically solving equation 2 with an RK23 ODE solver in Python. The density matrix is sampled at several time steps and used to calculate the current in the leads for each time step. In conclusion, the process begins with an input function of time, resulting in a current represented as a time series with discrete points in time.

We can illustrate the dynamics in the quantum dot reservoir, by analyzing the time dependent density matrix. In the QRC package, the module "*Reservoir_dynamics*" contains functions to illustrate the energy and charge distribution in the system. See figure 4 for an example of how the charge is distributed in the system as a function of time when the input is a sine wave.

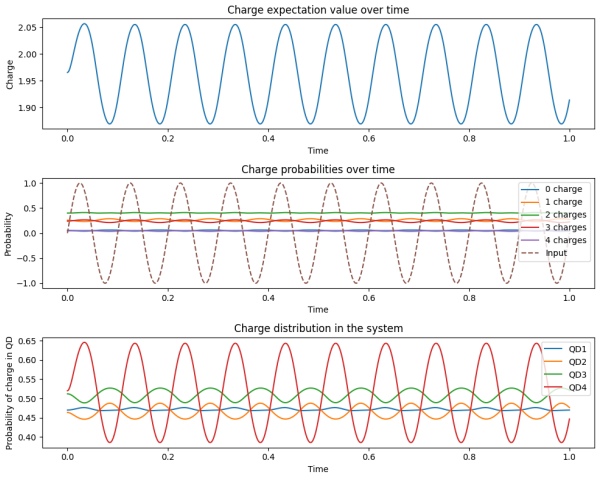

Figure 4: Figures showing charge distribution as function of tiome when input is a sine wave. Top figure shows charge expectation value over time. Middle figures shows input function and probability for different number of charges in the system. Bottom figure shows how the charge is distributed in the reservoir.

4 Training the network

The first step of training the network is defining an input function for the reservoir and a corresponding target function. These functions are then divided into three parts, "warmup", "training" and "test", as illustrated with the input function in figure 5. The reason for the "warmup" part is to make the reservoir independent of the initial conditions of the reservoir. Because of this, the output from the reservoir during warmup is discarded.

In the training step, the task is to find weights of the linear layer that minimize the mean squared error ($\frac{1}{L}\sum_{k=1}^{L}(y_{pred} - y_{targ})^2$ where L is the number of time steps) between the prediction (y_{pred}) and the target(y_{targ}). Since only one layer has to be trained, it is a computationally simple task to find optimal weights. In our case, we use a finished function in the Python package reservoirpy [4] to optimize the weights. We also add a regression term to avoid overfitting to the training data. For details about how this optimization can be done by finding the Moore-Penrose pseudoinverse matrix for the input features, see for example [5].

In the test step, the output from the reservoir is sent through the trained neural network and the performance is measured by comparing the predictions and targets. The performance can, for example, be measured in normalized mean squared error:

$$\frac{\frac{1}{L}\sum_{k=1}^{L}(y_{pred} - y_{targ})^2}{\sigma^2(\mathbf{y}_{targ})} \tag{4}$$

or in the following measurement that we call memory accuracy:

$$\frac{cov^2(\mathbf{y}_{targ}, \mathbf{y}_{pred})}{\sigma^2(\mathbf{y}_{targ})\sigma^2(\mathbf{y}_{pred})} \tag{5}$$

where σ^2 is the variance.

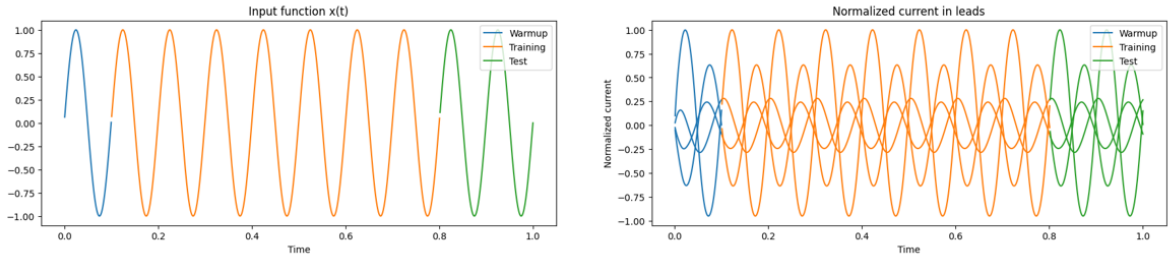

Figure 5: An example of how the input is split into warmup, training and test data. The input function generates current in the leads and is used to train and test the reservoir.

There is also a possibility to use "time multiplexing" when using the reservoir. This means, that we sample the current multiple times for each time step, by dividing each time step into smaller steps. The sampled currents are then saved in virtual nodes. When feeding the current from the reservoir into the network, the current from all virtual nodes is included. This means that if we sample the current V times for each time step (V virtual nodes), the number of weights in the readout layer will be given by $4 \cdot V$

5 Benchmark tests

To measure the reservoirs capability to store and encode data, we perform benchmark tests. The benchmark tests we use, are common to use for evaluation of quantum reservoir computing models, see for example [5] where the same tests have been used.

To test linear memory of the reservoir, which is the capacity to remember a certain input in the past, we employ the Timing Task. To test non-linear memory, which is the ability to remember multiple different inputs in the past and being able to do a non-linear transformation of these, we employ the Nonlinear Auto-Regressive Moving Average (NARMA) task. We also test the reservoir's ability to predict a chaotic time series by predicting the Mackey-Glass time series with the reservoir. To test the reservoir for different tasks, the module "$Task\_engine$" in the QRC package can be used.

5.1 Timing Task

In the timing task, the reservoir is given a step function as input and the task is to output a peak at a specified time after the step. To succeed with the task, the reservoir must remember the step of the input when it is supposed to peak. If the reservoir has decayed to the steady state before the timer time, it will fail.

Both when training and testing, a step function is given as input to the reservoir. Because the reservoir's input-to-output behavior is deterministic, the same reservoir output (current) will be used for both training and test. It is therefore important to notice that no generalization ability is tested in this task.

In figure 6 an example of the timing task with different timers (the time after the step the prediction should peak) is illustrated.

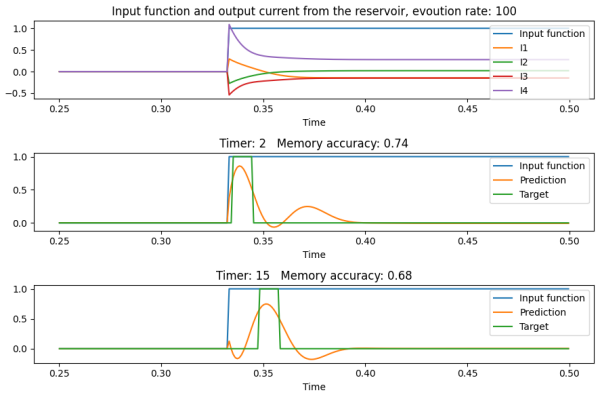

Figure 6: Example of the input and current for a random reservoir with the timing task and example of timing task with timer 2 and 15.

In figure 7 the performance averaged over 20 reservoirs is plotted against the timer time. This is done for evolution rates 100 and 200. From the plot, it is clear to see that the reservoir remembers for longer when the evolution rate is smaller. We can also see that the performance is better for shorter timers when the evolution rate is larger.

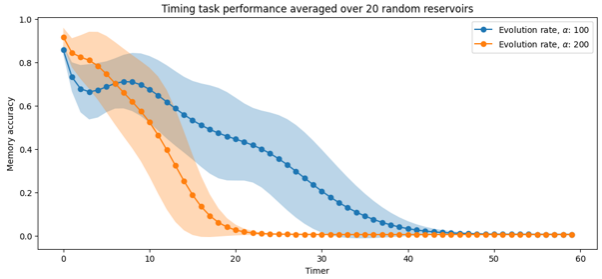

Figure 7: Performance of timing tasks averaged over 20 random reservoirs with evolution rate 100 and 200. The shaded region corresponds to the standard deviation of the memory accuracy.

By plotting the performance against timers for the same reservoir (see figure 8) we can see that the timer when the memory accuracy becomes 0, is inversely proportional to the evolution rate. From the figure, we can also see that the performance depending on timer time has an oscillating pattern. If this is because the memory is oscillating or if it is caused by another reason, is currently not known.

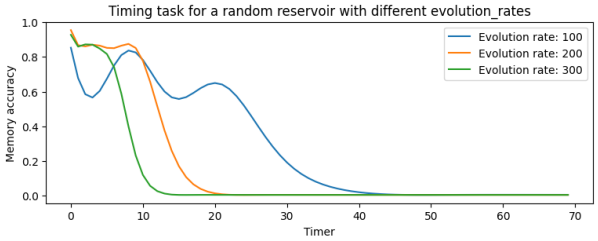

Figure 8: Timing task performance for one random reservoir with evolution rate 100, 200 and 300.

5.2 NARMA

To test the nonlinear memory of the reservoir, we employ the NARMA task. The task is to estimate a system y_k described by equation 7 for a random input function s_k. In this example, we randomize the input by summing 1000 random sine waves (equation 6), other options are for example to use white noise. The task can be varied by changing the value of n. To succeed with this task, the reservoir needs to make a nonlinear transformation depending on multiple past inputs.

$$s_k = \sum_{n=j}^{1000} a_j \sin\left(b_j t + c_j\right) \tag{6}$$

$$y_{k+1} = 0.3y_k + 0.01y_k \left(\sum_{j=0}^{n-1} y_{k-j} \right) + 1.5 s_{k-n+1} s_k + 0.1 \tag{7}$$

An example of how a random reservoir with evolution rate 150 is performing with the NARMA-5 (n=5) and NARMA-20 (n=20) task is illustrated in figure 9. To see how the performance depends on the evolution rate, the average performance for 20 reservoirs was plotted against the evolution rate in figure 10. From the figure, we can see that the NARMA-5 task has high performance for high evolution rates. This is reasonable because the reservoir only needs to remember a few steps in the past to successfully predict NARMA-5. For the NARMA-20 task, we can see that it has an optimum at around $\alpha = 100$.

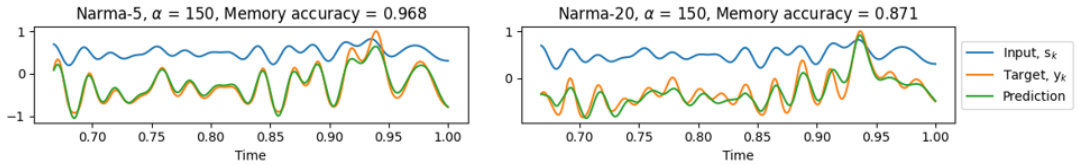

Figure 9: Perforance of the NARMA-5 and NARMA-20 task for a random reservoir with evolution rate 150.

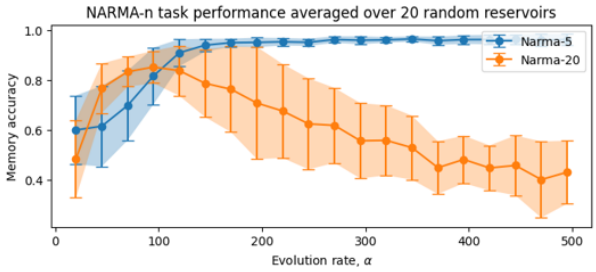

Figure 10: NARMA-5 and NARMA-20 task performance measured in memory accuracy against evolution rate averaged over 20 random reservoirs. Error bars corresponding to standard deviation are included.

5.3 Prediction of the Mackey-Glass Time Series

The Mackey Glass time series, is a time series commonly used in mathematical biology, described by the equation:

$$\frac{dP(t)}{dt} = \frac{aP(t-\tau)}{1+P(t-\tau)^n} - bP(t) \tag{8}$$

where a = 0.2, b = 0.1, n = 10. The higher τ is, the more chaotic the time series is, we use $\tau = 17$.

In this task, we test if the reservoir can predict the Mackey-Glass time series several time steps in the future. We use offline learning, meaning that the reservoir is fed with the ground truth time series for all time steps. For each time step, it predicts the time series a number of time steps in the future. It is later on fed with the true value. An option for future development of this project is to implement online learning, where the prediction is fed back into the reservoir.

In figure 11 examples of predicting 5 steps ahead and 50 steps ahead are illustrated. In figure 12, the average performance depending on the number of prediction steps for 10 random reservoirs is shown.

To optimize the prediction performance, the range that the reservoir parameters were initialized in was tuned and time multiplexing was used. In the figures 11 and 12, the reservoir parameters were set in the range $\epsilon_i \in [-0.1, 0.1]$, $t_{ij} \in [0, 10]$ and $U_{ij} \in [0, 0.2]$ and the input weights connecting the input function and lead chemical potentials set to $W_{in} = [-6, -3, 3, 12]$. The number of virtual nodes used for time multiplexing was set to $V = 5$.

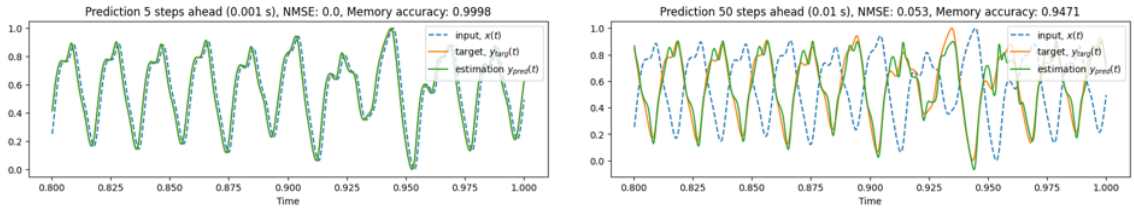

Figure 11: Example of prediction with 5 steps and 50 steps ahead of the Mackey-Glass time series.

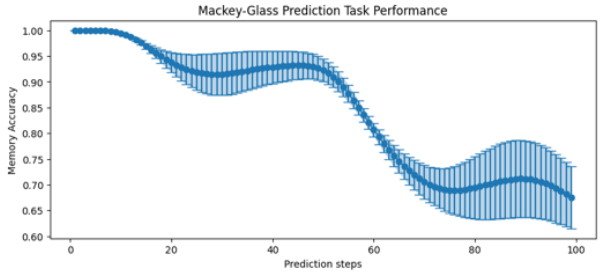

Figure 12: Performance in memory accuracy against the number of predicted steps. The performance is averaged over 10 random reservoirs. Error bars corresponding to standard deviation are included.

6 Conclusion and Outlook

In this project, we have been able to simulate a reservoir consisting of four coupled quantum dots connected to electronic leads. With the timing task and NARMA task, we have been able to test the reservoir's linear and non-linear memory. With the Mackey-Glass time series task, we were able to test the reservoir's ability to predict a chaotic time series.

Future directions for this project could be investigating how the physical properties of the reservoir affects the performance and how one can tune the reservoir for optimal performance. Other directions are for example implementing online learning, where output from the network is fed back into the reservoir.

References

[1] Kohei Nakajima. "Physical reservoir computing—an introductory perspective". In: *Japanese Journal of Applied Physics* 59.6 (May 2020), p. 060501. ISSN: 1347-4065. DOI: 10.35848/1347-4065/ab8d4f. URL: http://dx.doi.org/10.35848/1347-4065/ab8d4f.

[2] Matteo Cucchi et al. "Hands-on reservoir computing: a tutorial for practical implementation". In: *Neuromorphic Computing and Engineering* 2 (Aug. 2022). DOI: 10.1088/2634-4386/ac7db7.

[3] Gediminas Kiršanskas et al. “QmeQ 1.0: An open-source Python package for calculations of transport through quantum dot devices”. In: *Computer Physics Communications* 221 (Dec. 2017), pp. 317–342. ISSN: 0010-4655. DOI: 10.1016/j.cpc.2017.07.024. URL: http://dx.doi.org/10.1016/j.cpc.2017.07.024.

[4] Nathan Trouvain et al. “ReservoirPy: An Efficient and User-Friendly Library to Design Echo State Networks”. In: *Artificial Neural Networks and Machine Learning – ICANN 2020*. Ed. by Igor Farkaš, Paolo Masulli, and Stefan Wermter. Cham: Springer International Publishing, 2020, pp. 494–505. ISBN: 978-3-030-61616-8.

[5] Keisuke Fujii and Kohei Nakajima. “Harnessing disordered quantum dynamics for machine learning”. In: *Physical Review Applied* 8 (Feb. 2016). DOI: 10.1103/PhysRevApplied.8.024030.