Reservoir computing

Karol Bednarz

September 14, 2025

Contents

1 Introduction

1.1 Principles of reservoir computing

The state of reservoir dynamics can be expressed as:

$$h_{\mathrm{t}} = f\big((1-k)\cdot u_{\mathrm{t}}\cdot W_{\mathrm{in}} + k\cdot h_{\mathrm{t-1}}\cdot W_{\mathrm{h}} + y_{\mathrm{t-1}}\cdot W_{\mathrm{1b}} + b\big) \tag{1}$$

Where:

$h_{\mathrm{t-1}}$ – are the reservoir state respectively, from the previous time step,
u_{t} – is the observed data at time step t,
$y_{\mathrm{t-1}}$ – is the the predicted output state $t-1$,
$W_{\mathrm{in}} \in \mathbb{R}^{N_{\mathrm{u}}\times N_{\mathrm{h}}}$ – is the input weight matrix,
$W_{\mathrm{h}} \in \mathbb{R}^{N_{\mathrm{h}}\times N_{\mathrm{h}}}$ – is the internal weight matrix,
$W_{\mathrm{1b}} \in \mathbb{R}^{N_{\mathrm{h}}\times N_{\mathrm{y}}}$ – is the output fedback weight matrix,
$b \in \mathbb{R}^{N_{\mathrm{h}}}$ – is the bias vector.
f – is the activation function, typically tanh or sigmoid,
k – is the leaking rate, typically $k \in [0.1, 0.3]$.

With the computed reservoir dynamics, the output can be then obtained by:

$$y_{\mathrm{t}} = h_{\mathrm{t}}\cdot W_{\mathrm{out}} \tag{2}$$

Where:

$W_{\mathrm{out}} \in \mathbb{R}^{N_{\mathrm{h}}\times N_{\mathrm{y}}}$ – is the output weight matrix.

1.2 Echo state property

Any system that changes in a nonlinear way can work as the reservoir. However, starting a nonlinear system with random connection strengths creates problems. The reservoir is a system that feeds its outputs back into itself. This can make it unstable if the connection strengths aren't set up correctly. For example, if the internal connections are too strong, the system might get stuck giving the same output regardless of what input it receives. The random connection strengths must be chosen so the system doesn't grow out of control. For the system to work well, it must follow something called the "echo state property." This rule ensures that the reservoir's behavior eventually depends on the input signal rather than just its starting conditions. To meet this requirement, the internal connection matrix W_{h} is first set up using random values between -1 and 1. This matrix is then adjusted one time according to the echo state property rule:

$$W'_{\mathrm{h}} = \alpha \odot W_{\mathrm{h}} \tag{3}$$

$$W^{\dagger}_{\mathrm{h}} = \frac{\rho W_{\mathrm{h}}}{|\lambda_{\max}(W_{\mathrm{h}})|} \tag{4}$$

Where:

$\rho \in (0, 1)$ – is the spectral radius, typically $\rho \in [0.9, 1]$
$\lambda_{\max}(W_{\mathrm{h}})$ – is the largest eigenvalue of W_{h}.
$\alpha \in (0, 1)$ – is the sparsity coefficient, typically $\alpha \in [0.1, 0.3]$.

The spectral radius is a parameter that determines the amount of nonlinear interaction of input components through time.

Due to the recursive nature of the reservoir layer, such dynamics reflect trajectories of the past historical inputthe short-term memory (known as the fading memory). As another critical property for computing the RC principle, short-term memory can be quantitative measured by the coefficient of memory capacity

$$MC = \sum_{\mathrm{k}=1}^{\infty} MC_{\mathrm{k}} = \sum_{\mathrm{k}=1}^{\infty} d^2(u_{\mathrm{t}-\mathrm{k}}, y_{\mathrm{t}}) = \sum_{\mathrm{k}=1}^{\infty} \frac{\mathrm{cov}^2(u_{\mathrm{t}-\mathrm{k}}, y_{\mathrm{t}})}{\sigma^2(u_{\mathrm{t}})\sigma^2(y_{\mathrm{t}})} \tag{5}$$

Where:

$d^2(u_{\mathrm{t}-\mathrm{k}}, y_{\mathrm{t}})$ – is the square of the correlation coefficient between the output y_{t} and the input $u_{\mathrm{t}-\mathrm{k}}$ with a delay of k time steps,

According to the Lyapunov stability analysis, a large memory capacity is needed to compute the RC principle, which can be achieved at the asymptotically stable region.

2 Learning algorithm

The training of the reservoir computing model involves adjusting only the output weights W_{out}. The input weights W_{in}, internal weights W_{h}, and feedback weights $W_{1\mathrm{b}}$ are typically initialized randomly and remain fixed during training. The training process can be summarized in the following steps:

$$Y = H \cdot W_{\mathrm{out}} \tag{6}$$

Where:

$Y \in \mathbb{R}^{T \times N_{\mathrm{y}}}$ – is the matrix of target outputs for all time steps,

$H \in \mathbb{R}^{T \times N_\mathrm{h}}$ – is the matrix of reservoir states for all time steps,
T – is the total number of time steps.

In general, the W_out can be directly obtained by calculating the Moore-Penrose pseudoinverse of the reservoir states matrix H with respect to the target outputs matrix Y:

$$W_\mathrm{out} = Y \cdot H^\dagger \cdot (H \cdot H^\dagger + \eta I)^{-1} \tag{7}$$

Where:

$H^\dagger$ – is the Moore-Penrose pseudoinverse of matrix H,
η – is the regularization parameter, typically $\eta \in [10^{-6}, 10^{-2}]$,
I – is the identity matrix of size $N_\mathrm{h} \times N_\mathrm{h}$.

3 Sample implementation in Python

```python
import matplotlib.pyplot as plt
import numpy as np
import torch
import torch.nn as nn
from scipy.integrate import solve_ivp


class LorenzESN(nn.Module):
    """
    Echo State Network with Lorenz system dynamics in the reservoir.

    The reservoir states evolve according to modified Lorenz equations,
    providing rich chaotic dynamics for temporal pattern learning.
    """

    def __init__(
        self,
        input_size,
        reservoir_size,
        output_size,
        spectral_radius=0.9,
        input_scaling=1.0,
        leaking_rate=1.0,
        lorenz_coupling=0.1,
        sigma=10.0,
        rho=28.0,
        beta=8.0 / 3.0,
    ):
        """
        Args:
            input_size: Dimension of input
            reservoir_size: Size of reservoir (should be multiple of 3 for Lorenz components)
            output_size: Dimension of output
            spectral_radius: Spectral radius of reservoir weight matrix
            input_scaling: Scaling factor for input weights
            leaking_rate: Leaking rate for reservoir updates
            lorenz_coupling: Coupling strength between Lorenz subsystems
            sigma, rho, beta: Lorenz system parameters
        """
        super(LorenzESN, self).__init__()

        self.input_size = input_size
        self.reservoir_size = reservoir_size
        self.output_size = output_size
        self.leaking_rate = leaking_rate
        self.lorenz_coupling = lorenz_coupling

        # Lorenz parameters
        self.sigma = sigma
        self.rho = rho
        self.beta = beta

        # Ensure reservoir size is multiple of 3 for Lorenz triplets
        self.num_lorenz_systems = reservoir_size // 3
        self.actual_reservoir_size = self.num_lorenz_systems * 3

        # Initialize input weights
        self.W_in = nn.Parameter(
```

```
            torch.randn(self.actual_reservoir_size, input_size) * input_scaling,
            requires_grad=False,
        )

        # Initialize reservoir coupling weights (sparse connectivity between Lorenz systems)
        W_res = torch.randn(self.actual_reservoir_size, self.actual_reservoir_size)
        W_res = self.make_sparse(W_res, sparsity=0.1)  # 10% connectivity

        # Scale to desired spectral radius
        eigenvals = torch.linalg.eigvals(W_res)
        current_spectral_radius = torch.max(torch.abs(eigenvals)).item()
        W_res = W_res * (spectral_radius / current_spectral_radius)

        self.W_res = nn.Parameter(W_res, requires_grad=False)

        # Output weights (trainable)
        self.W_out = nn.Linear(self.actual_reservoir_size, output_size)

        # Initialize reservoir state
        self.register_buffer("reservoir_state", torch.zeros(self.actual_reservoir_size))

    def make_sparse(self, matrix, sparsity=0.1):
        """Make matrix sparse by randomly setting elements to zero"""
        mask = torch.rand_like(matrix) < sparsity
        return matrix * mask.float()

    def lorenz_derivatives(self, state):
        """Compute Lorenz derivatives for the entire reservoir state"""
        # Reshape to (num_systems, 3) for easier processing
        lorenz_state = state.view(self.num_lorenz_systems, 3)

        # Compute Lorenz derivatives for each system
        x, y, z = lorenz_state[:, 0], lorenz_state[:, 1], lorenz_state[:, 2]

        dx_dt = self.sigma * (y - x)
        dy_dt = x * (self.rho - z) - y
        dz_dt = x * y - self.beta * z

        # Stack derivatives
        derivatives = torch.stack([dx_dt, dy_dt, dz_dt], dim=1)

        return derivatives.view(-1)  # Flatten back to 1D

    def lorenz_dynamics(self, state, dt=0.001):
        """
        Apply Lorenz dynamics to reservoir state.
        State is organized as [x1,y1,z1, x2,y2,z2, ..., xN,yN,zN]
        """
        # TODO: Apply RK4 integration instead of Euler
        k1 = dt * self.lorenz_derivatives(state)
        k2 = dt * self.lorenz_derivatives(state + 0.5 * k1)
        k3 = dt * self.lorenz_derivatives(state + 0.5 * k2)
        k4 = dt * self.lorenz_derivatives(state + k3)
        new_state = state + (k1 + 2 * k2 + 2 * k3 + k4) / 6
        return new_state

    def forward(self, input_sequence):
        """
        Forward pass through the ESN.

        Args:
```

```
            input_sequence: (seq_len, batch_size, input_size) or (seq_len, input_size)

        Returns:
            outputs: (seq_len, batch_size, output_size) or (seq_len, output_size)
        """
        if input_sequence.dim() == 2:
            input_sequence = input_sequence.unsqueeze(1)  # Add batch dimension
            squeeze_output = True
        else:
            squeeze_output = False

        seq_len, batch_size, _ = input_sequence.shape

        # Initialize states for batch
        reservoir_states = self.reservoir_state.unsqueeze(0).repeat(batch_size, 1)
        all_states = []

        for t in range(seq_len):
            # Current input
            current_input = input_sequence[t]  # (batch_size, input_size)

            for b in range(batch_size):
                # Apply Lorenz dynamics
                reservoir_states[b] = self.lorenz_dynamics(reservoir_states[b])

                # Add input and reservoir coupling
                input_contribution = torch.matmul(self.W_in, current_input[b])
                reservoir_contribution = torch.matmul(self.W_res, reservoir_states[b])

                # Leaky integration
                new_state = (1 - self.leaking_rate) * reservoir_states[
                    b
                ] + self.leaking_rate * torch.tanh(
                    input_contribution + self.lorenz_coupling * reservoir_contribution
                )

                reservoir_states[b] = new_state

            all_states.append(reservoir_states.clone())

        # Stack all states: (seq_len, batch_size, reservoir_size)
        all_states = torch.stack(all_states)

        # Compute outputs
        outputs = self.W_out(all_states)

        if squeeze_output:
            outputs = outputs.squeeze(1)

        return outputs

    def reset_state(self):
        """Reset reservoir state"""
        self.reservoir_state.zero_()


def generate_lorenz_data(num_steps=1000, dt=0.01, sigma=10.0, rho=28, beta=8.0 / 3.0):
    """Generate Lorenz attractor data for testing"""

    def lorenz(t, state):
        x, y, z = state
        return [sigma * (y - x), x * (rho - z) - y, x * y - beta * z]
```

```
    t_span = (0, num_steps * dt)
    t_eval = np.arange(0, num_steps * dt, dt)
    initial_state = [0.1, 0.1, 0.1]

    sol = solve_ivp(lorenz, t_span, initial_state, t_eval=t_eval, method="DOP853")
    return torch.tensor(sol.y.T, dtype=torch.float32)  # Shape: (num_steps, 3)


def train_esn_example():
    """Example training script"""

    # Generate synthetic data (predicting next step of Lorenz system)
    data = generate_lorenz_data(num_steps=2000, dt=0.01)
    # data =

    # Prepare sequences
    seq_length = 100
    X = []
    y = []

    for i in range(len(data) - seq_length):
        X.append(data[i : i + seq_length])
        y.append(data[i + 1 : i + seq_length + 1])

    X = torch.stack(X)  # (num_sequences, seq_length, 3)
    y = torch.stack(y)  # (num_sequences, seq_length, 3)

    # Split data
    train_size = int(0.8 * len(X))
    X_train, X_test = X[:train_size], X[train_size:]
    y_train, y_test = y[:train_size], y[train_size:]

    # Create ESN
    esn = LorenzESN(
        input_size=3,
        reservoir_size=500 * 3,  # 100 Lorenz systems
        output_size=3,
        spectral_radius=0.95,
        input_scaling=1.0,
        leaking_rate=0.1,
        lorenz_coupling=0.05,
    )

    print("Training ESN with Lorenz reservoir dynamics...")

    # Training (only train output weights using ridge regression)
    esn.eval()  # Set to eval mode to disable gradient computation for reservoir

    # Collect reservoir states for all training sequences
    all_reservoir_states = []
    all_targets = []

    with torch.no_grad():
        for i in range(len(X_train)):
            esn.reset_state()
            states = esn.forward(X_train[i])  # Don't use outputs, just collect states

            # Get intermediate states from the reservoir
            reservoir_states = []
            esn.reset_state()
            for t in range(X_train[i].shape[0]):
```

```
                # Manual forward pass to collect states
                current_input = X_train[i][t]
                esn.reservoir_state = esn.lorenz_dynamics(esn.reservoir_state)

                input_contribution = torch.matmul(esn.W_in, current_input)
                reservoir_contribution = torch.matmul(esn.W_res, esn.reservoir_state)

                new_state = (
                    1 - esn.leaking_rate
                ) * esn.reservoir_state + esn.leaking_rate * torch.tanh(
                    input_contribution + esn.lorenz_coupling * reservoir_contribution
                )

                esn.reservoir_state = new_state
                reservoir_states.append(esn.reservoir_state.clone())

            reservoir_states = torch.stack(reservoir_states)
            all_reservoir_states.append(reservoir_states)
            all_targets.append(y_train[i])

    # Concatenate all data
    X_reservoir = torch.cat(
        all_reservoir_states, dim=0
    )  # (total_timesteps, reservoir_size)
    y_flat = torch.cat(all_targets, dim=0)  # (total_timesteps, 3)

    # Ridge regression for output weights
    ridge_param = 1e-6
    I = torch.eye(X_reservoir.shape[1])

    # Solve: W_out = (X^T X + I)^{-1} X^T y
    XTX = torch.matmul(X_reservoir.T, X_reservoir)
    XTy = torch.matmul(X_reservoir.T, y_flat)
    W_out_optimal = torch.linalg.solve(XTX + ridge_param * I, XTy)

    # Set the optimal weights
    esn.W_out.weight.data = W_out_optimal.T
    esn.W_out.bias.data.zero_()

    # Test the model
    test_predictions = []
    test_targets = []

    with torch.no_grad():
        for i in range(min(5, len(X_test))):  # Test on first 5 sequences
            esn.reset_state()
            pred = esn.forward(X_test[i])
            test_predictions.append(pred.numpy())
            test_targets.append(y_test[i].numpy())

    # Calculate MSE
    mse = np.mean(
        [(pred - target) ** 2 for pred, target in zip(test_predictions, test_targets)]
    )
    print(f"Test MSE: {mse:.6f}")

    return esn, test_predictions, test_targets, data.numpy()


if __name__ == "__main__":
    # Run example
    esn, predictions, targets, original_data = train_esn_example()
```

```
    # Plot results
    plt.figure(figsize=(15, 10))

    # Plot original Lorenz attractor
    plt.subplot(2, 3, 1)
    plt.plot(original_data[:1000, 0], original_data[:1000, 2])
    plt.title("Original Lorenz Attractor (X-Z plane)")
    plt.xlabel("X")
    plt.ylabel("Z")

    # Plot prediction vs target for first test sequence
    if predictions:
        pred = predictions[0]
        target = targets[0]

        plt.subplot(2, 3, 2)
        plt.plot(target[:, 0], "b-", label="Target X", alpha=0.7)
        plt.plot(pred[:, 0], "r--", label="Predicted X", alpha=0.7)
        plt.title("X Component Prediction")
        plt.legend()

        plt.subplot(2, 3, 3)
        plt.plot(target[:, 1], "b-", label="Target Y", alpha=0.7)
        plt.plot(pred[:, 1], "r--", label="Predicted Y", alpha=0.7)
        plt.title("Y Component Prediction")
        plt.legend()

        plt.subplot(2, 3, 4)
        plt.plot(target[:, 2], "b-", label="Target Z", alpha=0.7)
        plt.plot(pred[:, 2], "r--", label="Predicted Z", alpha=0.7)
        plt.title("Z Component Prediction")
        plt.legend()

        # 3D phase space comparison
        plt.subplot(2, 3, 5)
        plt.plot(target[:, 0], target[:, 2], "b-", label="Target", alpha=0.7)
        plt.plot(pred[:, 0], pred[:, 2], "r--", label="Predicted", alpha=0.7)
        plt.title("Phase Space (X-Z)")
        plt.xlabel("X")
        plt.ylabel("Z")
        plt.legend()

        # Error plot
        plt.subplot(2, 3, 6)
        error = np.abs(pred - target)
        plt.plot(error[:, 0], label="X error")
        plt.plot(error[:, 1], label="Y error")
        plt.plot(error[:, 2], label="Z error")
        plt.title("Absolute Error")
        plt.legend()

    plt.tight_layout()
    plt.show()
```