Increasing Reservoir Computer Responsiveness to Initial Conditions

July 8, 2021

Overview

Given an n_d dimensional orbit $\mathbf{u}(t)$ for $t_0 < t < T$, through a system that we will attempt to learn, a reservoir computer is trained by driving the reservoir with the input $\mathbf{u}(t)$, then projecting $\mathbf{u}(t)$ onto the resulting driven orbit of the reservoir nodes.

This occurs as follows. Given $\mathbf{u}(t)$, choose an n_r dimensional reservoir initial condition $\mathbf{r}_0$. Standard practice selects r_0 randomly, but it is more useful if $\mathbf{r}_0 = \phi(\mathbf{u}(t_0))$ for some ϕ. Then, we solve the following initial value problem numerically for solution values at the discrete time points $\{t_0, ..., tt_n\}$.

$$\frac{d\mathbf{r}}{dt} = -\gamma\big[\mathbf{r}(t) + f\big(A\mathbf{r}(t) + \sigma W_{\text{in}}\mathbf{u}(t)\big)\big], \quad \mathbf{r}(t_0) = \mathbf{r}_0 \tag{1}$$

Here $\gamma > 0$ and $\sigma > 0$ are hyper parameters, f is an activation function W_{in} is an $n_r \times n_d$ dimensional read-in matrix and A is an adjacency matrix that governs how the nodes in the reservoir interact.

Let R represent the numerical solution to the IVP above and let U represent $\mathbf{u}(t)$ sampled at corresponding time points. That is,

$$R = \begin{bmatrix} \mathbf{r}(t_0)^T \\ \mathbf{r}(t_1)^T \\ \vdots \\ \mathbf{r}(t_{n-1})^T \\ \mathbf{r}(T)^T \end{bmatrix} \qquad U = \begin{bmatrix} \mathbf{u}(t_0)^T \\ \mathbf{u}(t_1)^T \\ \vdots \\ \mathbf{u}(t_{n-1})^T \\ \mathbf{u}(T)^T \end{bmatrix}$$

If we use $n + 1$ time values, then R is an $(n + 1) \times n_r$ matrix and U is an $(n + 1) \times n_d$ matrix.

We then approximate the projection of $\mathbf{u}(t)$ onto $\mathbf{r}(t)$ by finding a $n_d \times n_r$ matrix W_{out} that projects the rows of U onto the rows of R.

This is computed by solving the Tikanov regression problem

$$||W_{\text{out}}R^T - U||_2 + \alpha||W_{\text{out}}||$$

for minimizer W_{out}.

It is important to note that W_{out} does not send $\mathbf{r}(t)$ to $\mathbf{u}(t)$. Instead attempts to send every row of R to the corresponding row in U . Therefore, the usefulness of a particular W_{out} for a given problem depends on the kind of data that is in R and U. By adjusting the data (adding or removing rows) in R and U we can provide additional conditions or constraints to our desired W_{out}.

Since we are interested in the problem of arbitrary initial conditions, we note that in standard reservoir computer training, only a single row of R and U corresponds to an initial condition. We hypothesize that the importance of W_{out} getting this projection correct relative to the all the other rows, is not very high. This results in misaligned predictions. More explicitly, for initial test data $\hat{\mathbf{u}}_0$ and $\hat{\mathbf{r}}_0 = \phi(\hat{\mathbf{u}}_0)$ the magnitude of $||W_{\text{out}}\hat{\mathbf{r}}_0 - \hat{\mathbf{u}}_0||$ is large. If the reservoir is unable to match the initial test data even before solving the autonomous IVP, we can't expect it to predict accurately.

We attempt to remedy this by teaching W_{out} to associate multiple initial conditions from the training data with corresponding reservoir initial conditions. In theory, we could just augment R and U with initial condition mappings like this:

$$R_{\text{aug}} = \begin{bmatrix} R \\ \phi(\mathbf{u}(\tau_1)) \\ \phi(\mathbf{u}(\tau_2)) \\ \vdots \\ \phi(\mathbf{u}(\tau_k)) \end{bmatrix} \qquad U_{\text{aug}} = \begin{bmatrix} U \\ \mathbf{u}(\tau_1) \\ \mathbf{u}(\tau_2) \\ \vdots \\ \mathbf{u}(\tau_k) \end{bmatrix}$$

where $\tau_1, ...\tau_k$ are random times in (t_0, T). In fact, this might do better than our current window training method!

However, your author believes the evolution of the reservoir computer state immediately after a new initial condition is important to include in R and U. This is done with the intention of teaching W_{out} to map the beginnings of orbits to the correct places rather than just sending the initial condition to the right place. Thus, we use windows over the training data in the current algorithm.

Initial Condition Mappings

The initial condition mapping, ϕ, is a key element of creating a correspondence between an orbit in the training signal space and an orbit in the reservoir space. By reservoir space, we mean the n_r dimensional space that contains reservoir node trajectories.

If we train a reservoir with standard techniques, it can continue the trajectory of the training orbit. Using the notation from before, this is because

$$\mathbf{u}(T) \approx W_{\text{out}}\mathbf{r}(T)$$

which only occurs because $\mathbf{u}(T)$ and $\mathbf{r}(T)$ were included in the matrices used to solve for W_{out}. In other words, W_{out} "learned" what reservoir initial state should correspond to $\mathbf{u}(T)$ because W_{out} is the solution to a regularized least squares problem that creates a correspondence between $\mathbf{u}(T)$ and $\mathbf{r}(T)$.

However, what if we want to know the evolution of our unknown system from a new initial condition $\mathbf{u}^\star$ and we don't have any orbit leading up to $\mathbf{u}^\star$ to use to train our reservoir? Our trained reservoir is an n_r dimensional ODE that requires an initial condition in $\mathbb{R}^{n_r}$. If we assume our reservoir has indeed "learned" the unknown system, there should be an n_r dimensional initial condition $\mathbf{r}^\star$ such that the evolution of the reservoir nodes from this point corresponds to the evolution of the unknown system from $\mathbf{u}^\star$.

We know that W_{out} sends reservoir node states to the training signal space, and here we are trying to do the inverse operation, send a point in the training signal space to the reservoir space. Thus, a natural choice for mapping $\mathbf{u}^\star$ to an appropriate $\mathbf{r}^\star$ is the pseudo inverse $W_{\text{out}}^\dagger$. However, in practice, the pseudo inverse does not work. This appears to be because it does not respect the bounds on reservoir node state magnitude that are imposed by the activation function. If $f(\mathbf{r}) = \tanh(\mathbf{r})$, reservoir node states will remain within $[-1, 1]$ for all time (after a transient period where they travel to this interval). Thus, the reservoir node states that are used to create W_{out} are all within this interval, and therefore, the learned dynamics of the unknown system are embedded in the reservoir space inside the hyper-cube $[-1, 1]^{n_r}$. Thus, the important reservoir initial conditions are likely to all lie within this hyper-cube. However, in practice the pseudo inverse does not respect this region, and it's proposed initial condition vectors often end up in the transient region outside of this hyper cube. This leads to predictions that begin outside the of the area of interest, and then converge to the expected dynamics after a short time.

From this experience we hypothesize that good initial condition mappings probably place the reservoir condition inside the constraining hyper cube.

Inverting the Initial Condition Mapping

When we discussed using the pseudo inverse, we assumed that W_{out} was already known and we used $W_{\text{out}}^\dagger$ to create an initial condition mapping. However this method did not work. Instead of building an initial condition mapping from W_{out}, we take a different approach. We chose an arbitrary mapping $\phi(\mathbf{u}) = \mathbf{r}$ from the training space to the reservoir space and train W_{out} to invert $\phi(\mathbf{u})$. This is done by adding many initial condition correspondences $\{(\phi(\mathbf{u}_i), \mathbf{u}_i)\}_{i=1}^k$ to the training data as explained previously.

When this process is complete,

$$W_{\text{out}}\phi(\mathbf{u}^\star) \approx \mathbf{u}^\star$$

for any $\mathbf{u}^\star$.

Because of this we can use $\mathbf{r}^\star = \phi(mathbfu^\star)$ as an initial condition for the trained reservoir and be confident that the beginning of the reservoir orbit will align with the beginning of the orbit though unknown system.

This occurs because when we generate a trained reservoir orbit $\hat{\mathbf{r}}(t)$ and apply W_{out} to produce a prediction,

$$\hat{\mathbf{u}}(t) = W_{\text{out}}\hat{\mathbf{r}}(t)$$

we know that

$$\hat{\mathbf{u}}(t_0) = \mathbf{u}^\star \approx W_{\text{out}}\phi(\mathbf{u}^\star) = W_{\text{out}}\hat{\mathbf{r}}^\star = W_{\text{out}}\hat{\mathbf{r}}(t_0)$$

so the beginning of the orbits will correspond.

The Role of Transience

In untrained reservoir computers, the dynamical system always has at least one attracting fixed point. The location of this fixed point depends on the value of the driving signal. If the reservoir initial condition is not exactly this attracting fixed point, the orbit of the reservoir node will have an initial period of transience until it moves close enough to the attracting fixed point. After this transient period, the orbit of the node will simply follow the systems attracting fixed points as their location evolves in time.

We can set our unknown initial condition $\mathbf{r}^\star$ to the location of an attracting fixed point corresponding to the input $\mathbf{u}(t)$. This creates a reservoir node orbit with no transience. That is, upon supplying the initial condition, reservoir nodes will follow the oscillations of the fixed point without any period of moving "in range". We use this method in our research and call it the "relax" method. This is because, in order to compute the location of the attracting fixed point, we give the untrained reservoir the constant input $\mathbf{u}^\star$ and allow the reservoir nodes to relax into a steady state at the attracting fixed point.

However, preliminary experimentation showed that allowing some transience in the initial condition mapping was, in fact, helpful. To accommodate this, we use the function $\phi(\mathbf{u}) = f(W_{\text{in}}\mathbf{u})$. Here W_{in} sends the n_d dimensional initial condition to the reservoir space, $\mathbb{R}^{n_r}$, and applying the activation function f to the resulting n_r dimensional vector constrains the initial condition to the aforementioned hyper cube. The result will likely not be a reservoir computer fixed point, but it will be close to one. This provides a little bit of transience to the beginning of a node's orbit. We call this method "activf" in our research.

It remains to be seen which initial condition mapping is the best, but the fact that this method outperforms the "relax" method suggests that transience may be useful.

Algorithm Description

From initial condition $\mathbf{u}_0$, generate an array $(m \times n_d)$ of samples, U from the system you want to learn. Set the reservoir initial condition with $\mathbf{r}_0 = W_{\text{in}}\mathbf{u}_0$. Drive the reservoir states with the solution array to obtain R, the $(m \times n_r)$ array of reservoir node states.

Solve $||W_{\text{out}}R - U||_2$ for minimizer W_{out}. Tikanov regression with regularization parameter α gives

$$W_{\text{out}} = (R^TR - \alpha I)^{-1}R^TU$$

When n is large the cost of storing R in memory can be prohibitive. This problem is addressed by computing $R^T R$ and $R^T U$ in batches. We write

$$R = \begin{bmatrix} R_1 \\ R_2 \\ R_3 \\ \vdots \\ R_k \end{bmatrix}$$

where each Ri is an $(m_i \times n_r)$ array and $\sum_i^k m_i = m$.

Then

$$R^T R = \begin{bmatrix} R_1 & R_2 & R_3 & \cdots & R_k \end{bmatrix} \begin{bmatrix} R_1 \\ R_2 \\ R_3 \\ \vdots \\ R_k \end{bmatrix} = \sum_i^k R_i^T R_i$$

If we break up U into

$$U = \begin{bmatrix} U_1 \\ U_2 \\ \vdots \\ U_k \end{bmatrix}$$

where each U_i has dimension $(m_i \times n_d)$ we can compute $R^T U = \sum_i^k R_i^T U_i$. This allows us to compute W_{out} in batches. If the resulting matrices are saved, W_{out} can be updated with additional data.

A challenge with this approach is that the reservoir computer only associates one initial conditions with data. The even if the trained reservoir computer can continue the orbit on which it was trained, it may not learn to predict the orbit of an arbitrary initial condition, even if this new initial condition was close to the original initial condition. This paper presents a solution to this problem.

Continuing with the concept of batch computation of W_{out} we point out that there is no requirement that R_i corresponds with R_j when $j \neq i$. That is, whereas before, we assumed that R was a continuous stream of node states, and W_{out} projects R onto U, what is really true is that W_{out} attempts to send the rows of R to the associated rows in U. Therefore, the matrix U may contain orbits from multiple different initial conditions as long as the corresponding rows of R are the response of the reservoir computer to those orbits and their corresponding initial conditions.

Thus, if U_i contains a discretized solution to an ode, then the first row of R_i should be $W_{\text{in}}\mathbf{u}_0$ where u_0 is the first row of U_i and the remaining rows should be the evolution of node states with input from the entries of U_i. Therefore, each subsection of R must associate with U but the subsections R_i need not relate to each other.

This means that a reservoir computer may be trained with multiple different streams of input. Furthermore, on each individual stream, the reservoir computer can reset it's initial condition so that it learns to associate initial conditions with the appropriate response.

How to break up the data is an important question, because if the batch windows are too small, the reservoir computer will not be trained on long term prediction.

Luckily, since there is no limit on how many rows are in R, we can provide orbit association on multiple time scales if appropriate for the problem at hand. This is done by concatenating different length streams of input data into one large U matrix (Or computing the solution to the Tikhanov Regression problem in batches, with each input data matrix as a separate batch.

Next, focusing on a particular input stream, and assuming it is continuous, we can break it up into overlapping time windows and drive the reservoir computer with each window separately. For each window, we reset the reservoir internal initial condition to correspond with the first input condition of the particular time window. After this we can map the driven internal states back on to the concatenation of time windows. This allows us to train a reservoir computer to replicate an orbit starting from multiple places along the orbit. The number and size of time windows is a hyper parameter that can be tuned.

Algorithm Example

Next we will consider an example of applying this algorithm to learning an ODE. Let's assume we are using a discrete solver for the ODE and therefore can define a function F which accepts an array of m time values $\mathbf{t} = [t_1, t_2, ..., t_m]$ and an initial condition $\mathbf{u}$ where $\mathbf{u}$ is a vector of length n_d. Passing these values to F produces $U = F(t, \mathbf{u})$ where U is a $(m \times n_d)$ matrix and the i^{th} row of U corresponds to the solution of the ODE at time t_i. Similarly, we can define a function G that corresponds to the untrained reservoir ODE so that for a n_r dimensional vector $\mathbf{r}$, $R = G(t, \mathbf{r}, U)$ where R is an $(m \times n_r)$ matrix and the i^{th} row of R corresponds to the solution to the driven reservoir ode at time t_i.

Algorithm 1: Robust Reservoir Computer Training

```
Result: Write here the result
$\hat{R} \leftarrow (n_r \times n_r)$ array of zeros
$\hat{U} \leftarrow (n_r \times n_d)$ array of zeros
$T \leftarrow$ Maximum number of time-steps per batch
$\mathbf{v}_1, \mathbf{v}_2, \cdots \mathbf{v}_N \leftarrow$ Initial conditions for the ode
$\tau_1, \tau_2, \cdots \tau_k \leftarrow$ discretized time arrays for each initial condition
for $j$ in $1 \cdots N$ do
    $\mathbf{t} \leftarrow \tau_j$
    $\mathbf{u} \leftarrow \mathbf{v}_j$
    $U \leftarrow F(\mathbf{t}, \mathbf{u})$
    for $i$ in $1 \cdots k$ do
        start $\leftarrow T(i-1)+1$
        end $\leftarrow Ti$
        $U_i \leftarrow U$[start:end, : ]
        $\mathbf{t}_i \leftarrow \mathbf{t}$[start:end]
        $\mathbf{u}_i \leftarrow U_i[1, :]$
        $\mathbf{r}_0 \leftarrow W_{\text{in}} \mathbf{u}_i$
        $R_i \leftarrow G(\mathbf{t}_i, \mathbf{r}_0, U_i)$
        $\hat{R} \leftarrow \hat{R} + R_i^T R_i$
        $\hat{U} \leftarrow \hat{U} + R_i^T U_i$
    end
end
$W_{\text{out}} \leftarrow (\hat{R} - \alpha I)^{-1} \hat{U}$
```