Recurrent Neural Networks : a Dynamical Systems Perspective

Eric V. Jang
APMA136 Term Paper
Brown University
Providence, RI 02912
eric_jang@brown.edu

Abstract

Recurrent Neural Networks (RNNs) are artificial neural networks with feedback connections. The operation of an RNN generates a dynamic temporal state, which functions as an internal memory. Such architectures are well-suited for learning complex input-output sequences, such as generative models for speech, motor control, and time series prediction. Since RNNs incorporate state evolution over time, they can be analyzed as nonlinear dynamical systems. Understanding the phase portrait of an RNN may provide us with insight as to its memory capcity and stability of computation. This work implements a conjugate-gradient method for numerically finding fixed points in Echo State Networks, and analyzes the role of supervised learning on fixed points.

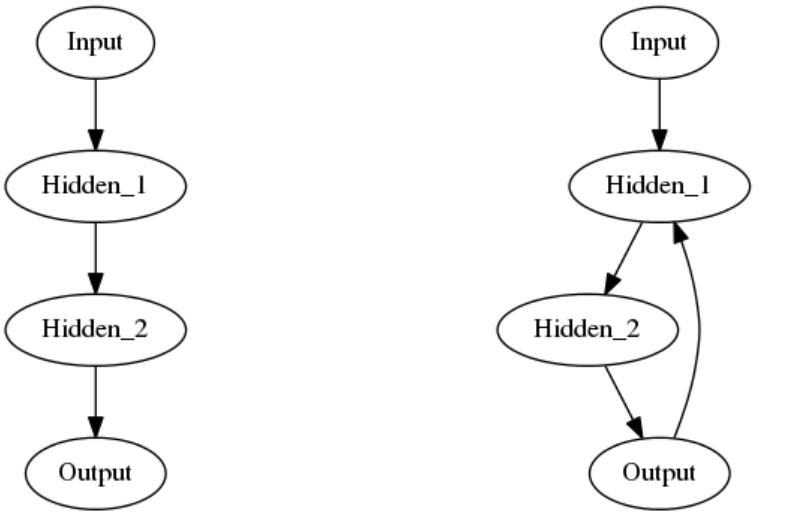
1 Introduction

Artificial Neural Networks (ANNs) are a family of neurally-inspired information processing systems that perform nonlinear transformations on high-dimensional inputs. Generally, an ANN is a collection of "neuronal units" that are simulated over time. Each unit integrates activations from its "presynaptic" units in order to update its own activation. The unit then forwards its activation to each of its "postsynaptic" neighbors, and the process starts over. Most ANN architectures satisfy the universal approximation theorem, and can perform arbitrary memory-bounded computation [1].

ANNs are particularly powerful because they can learn *computations* on data, whereas most statistics-based machine learning models match distributions to data. In most ANN architectures, units are organized into "layers" to impose connectivity structure between an otherwise fully-connected network. These layers can be chained together to form complex hierarchical representations of inputs. In feed-forward networks, layers are connected in a directed, acyclic graph. Activations are propagated sequentially along the graph until they terminate at the leaf nodes. In a Recurrent Neural Network (RNN), the connectivity between layers can form a directed multigraph. Through forward propagation, network activity can loop back and revisit the same layers over and over again. This creates an internal "memory" state that the network can then utilize to incorporate new observations with old ones.

Thanks to modern training techniques for neural networks, RNN models have shown promise in speech recognition, handwriting recognition, and content-addressable memory tasks [2, 3]. The term "Deep Learning" describes this trendy intersection of big data, neural networks, and high-performance computing.

Despite these improvements, it is not well understood how RNNs learn and implement their desired computations. How many feedforward layers are needed? How far back can the model remember? What exactly is being represented internally? Motivated by a general lack of understanding of *how* these "black box" models learn and compute, I set out to analyze RNNs from a dynamical systems



(a) Feedforward Neural Network (b) Recurrent Neural Network (RNN)

Figure 1: Example of two Artificial Neural Networks (ANNs)

perspective, as done in Sussillo & Barak2014 [4]. As an extension to their work, I also investigate the effects of training, activation functions, and transfer learning on the arrangement of fixed points in the system.

2 Notation

- Boldfaced symbols $\mathbf{x}$ denote a vector or a matrix.
- $\mathbf{A} \in \mathbb{R}^{\mathbf{M}\times\mathbf{N}}$ refers to a matrix A with M rows and N columns.
- Subscripts $\mathbf{F_i}$ denote the i'th element of the vector $\mathbf{F}$. Multiple subscripts in $\mathbf{M_{rc}}$ denote the value at row r, column c of matrix M.
- Superscripts are used to distinguish variables from each other (i.e. $\mathbf{W^{FB}}$ and $\mathbf{W^{o}}$ are both weight matrices)

3 Finding Fixed and Slow Points

Fixed points are often the first line of inquiry for characterizing the phase portrait of a system. Once found, analysis of linearized dynamics about these fixed points can reaveal how a local portion of the system behaves.

Let $\mathbf{x} \in \mathbb{R}^{\mathbf{N}}$ be a N-dimensional state vector, with dynamics governed by the following set of first-order, differential equations:

$$\dot{\mathbf{x}} = \frac{d\mathbf{F}}{dt} = \mathbf{F}(\mathbf{x}) \tag{1}$$

A fixed point $\mathbf{x}^\star \in \mathbb{R}^{\mathbf{N}}$ then satisfies the following:

$$\mathbf{F}(\mathbf{x}^\star) = \mathbf{0} \tag{2}$$

Consider the Taylor expansion of Equation (2) about the point $\mathbf{x} = \mathbf{x}^\star + \delta$:

$$\mathbf{F}(\mathbf{x}^\star + \delta) = \mathbf{F}(\mathbf{x}^\star) + \mathbf{F}'(\mathbf{x}^\star)\delta + \frac{\mathbf{1}}{\mathbf{2}}\mathbf{F}''(\mathbf{x}^\star)\delta^{\mathbf{2}} + ... \tag{3}$$

When δ is sufficiently small, the first-order terms of $F(\mathbf{x}^\star + \delta)$ dominate the higher-order terms in Equation (3), so $\mathbf{F}(\mathbf{x})$ can be approximated as a linear system nearby fixed points.

Let objective function $q \in \mathbb{R}$ be defined as:

$$q(\mathbf{x}) = \frac{\mathbf{1}}{\mathbf{2}}|\mathbf{F}(\mathbf{x})|^{\mathbf{2}} = \frac{\mathbf{1}}{\mathbf{2}}\sum_{\mathbf{i=1}}^{\mathbf{N}} \mathbf{F_i}(\mathbf{x})^{\mathbf{2}} \tag{4}$$

Observe that $q(\mathbf{x}) > \mathbf{0}$ for all $x \neq \mathbf{x}^\star$. By finding the $\mathbf{x}$ that minimizes this potential system, we can find the slowest regions of the system, whether it be slow or fixed points.

In this work, I chose the conjugate gradient (CG) method to minimize $q(\mathbf{x})$. It should be noted that gradient descent, which is a first-order optimization technique, typically arrives at the same minima as a second-order method (in practice). However, first-order descent occurs extremely slowly, because computation time is wasted repeatedly moving down the same direction, or oscillating back and forth across valleys in the energy surface. In contrast, the CG method picks a sequence of orthogonal search directions such that it jumps to the basin of a locally quadratic error surface after a fixed number of steps. This leads to faster convergence times.

One drawback of the CG method is that it requires the constant evaluation of the gradient $\nabla q(\mathbf{x})$ and Hessian $\mathbf{H}$ with respect to $\mathbf{x}$. For large systems this can become cumbersome; however, the examples dealt with in this work are tractable enough to evaluate the Jacobian and Hessian matrices in full.

To compute the gradient, we use Equation (4) and the chain rule:

$$\begin{aligned}
\nabla q(\mathbf{x})_\mathbf{i} &= \frac{\partial q}{\partial \mathbf{x_i}} && (5)\\
&= \frac{2}{2}\mathbf{F_1}(\mathbf{x})\frac{\partial \mathbf{F_1}(\mathbf{x})}{\partial \mathbf{x_i}} + \frac{2}{2}\mathbf{F_2}(\mathbf{x})\frac{\partial \mathbf{F_2}(\mathbf{x})}{\partial \mathbf{x_i}} + ... + \frac{2}{2}\mathbf{F_N}(\mathbf{x})\frac{\partial \mathbf{F_N}(\mathbf{x})}{\partial \mathbf{x_i}} && (6)\\
&= \dot{\mathbf{x_1}}\frac{\partial \mathbf{F_1}(\mathbf{x})}{\partial \mathbf{x_i}} + \dot{\mathbf{x_2}}\frac{\partial \mathbf{F_2}(\mathbf{x})}{\partial \mathbf{x_i}} + ...\dot{\mathbf{x_N}}\frac{\partial \mathbf{F_N}(\mathbf{x})}{\partial \mathbf{x_i}} && (7)\\
&= \frac{\partial \mathbf{F}(\mathbf{x})}{\partial \mathbf{x_i}}^\intercal \dot{x} && (8)
\end{aligned}$$

In vector notation, this can be re-written as

$$\nabla q(\mathbf{x}) = \mathbf{J}^\intercal \dot{\mathbf{x}} \tag{9}$$

The $\dot{\mathbf{x}}_k$ terms in the Hessian matrix vanish as the system approaches a fixed point, so it is sufficient to use the Gauss-Newton approximation to the Hessian matrix by ignoring the second term in 11.

$$\mathbf{H} = \begin{pmatrix} \frac{\partial^2 q}{\partial \mathbf{x_1}\partial \mathbf{x_1}} & \frac{\partial^2 q}{\partial \mathbf{x_1}\partial \mathbf{x_2}} & \cdots & \frac{\partial^2 q}{\partial \mathbf{x_1}\partial \mathbf{x_N}} \\ \frac{\partial^2 q}{\partial \mathbf{x_2}\partial \mathbf{x_1}} & \ddots & & \vdots \\ & & & \\ \frac{\partial^2 q}{\partial \mathbf{x_N}\partial \mathbf{x_1}} & \cdots & & \frac{\partial^2 q}{\partial \mathbf{x_N}\partial \mathbf{x_N}} \end{pmatrix} \tag{10}$$

$$\begin{aligned}
\frac{\partial^2 q}{\partial \mathbf{x_i}\partial \mathbf{x_j}} &= \sum_{k=1}^{N} \frac{\partial \mathbf{F_k}}{\partial \mathbf{x_i}}\frac{\partial \mathbf{F_k}}{\partial \mathbf{x_j}} + \sum_{k=1}^{N} \dot{\mathbf{x}}_k \frac{\partial \mathbf{F_k}}{\partial \mathbf{x_i}}\frac{\partial \mathbf{F_k}}{\partial \mathbf{x_j}} && (11)\\
&\approx \sum_{k=1}^{N} \frac{\partial \mathbf{F_k}}{\partial \mathbf{x_i}}\frac{\partial \mathbf{F_k}}{\partial \mathbf{x_j}} && (12)
\end{aligned}$$

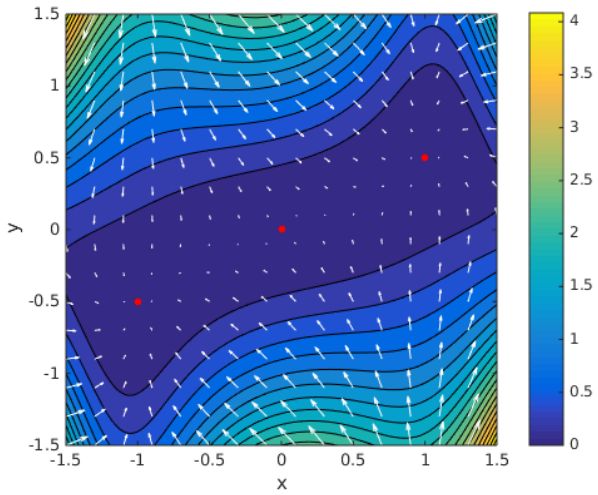


Figure 2: Contour plot of q as a function of x and y, with phase portrait of the system in Equations (14, 15). Fixed points found via $q(\mathbf{x})$ minimization are plotted in red. A saddle node at $(0, 0)$ funnels incoming trajectories towards either one of the stable, attracting nodes.

In vector notation, this is

$$\mathbf{H} = \mathbf{J}^\intercal \mathbf{J} \tag{13}$$

3.1 A Simple 2D Example

The following two-dimensional toy system illustrates the use of the $q(\mathbf{x})$ minimization technique for finding fixed points. Our system is given by:

$$\dot{x} = (1 - x^2)y \tag{14}$$

$$\dot{y} = \frac{x}{2} - y \tag{15}$$

Solving for x-nullclines, we have $\dot{x} = 0 \iff y = 0 \vee x = \pm 1$. Solving for the y-nullclines, we have $\dot{y} = 0 \iff y = \frac{1}{2}x$. The fixed points are located at the intersection of the x and y nullclines: $(1, \frac{1}{2})$, $(1, -\frac{1}{2})$, and $(0, 0)$.

The Jacobian is given by:

$$\mathbf{J}(\mathbf{x}, \mathbf{y}) = \begin{pmatrix} \frac{\partial f}{\partial x} & \frac{\partial f}{\partial y} \\ \frac{\partial g}{\partial x} & \frac{\partial g}{\partial y} \end{pmatrix} \tag{16}$$

$$= \begin{pmatrix} -2xy & 1 - x^2 \\ \frac{1}{2} & -1 \end{pmatrix} \tag{17}$$

Evaluating J at $(1, \frac{1}{2})$, $(1, -\frac{1}{2})$, and $(0, 0)$ and computing eigenvalues reveal a stable node, a stable node, and a saddle, respectively.

Figure 2 shows that results of $q(\mathbf{x})$ minimization are in agreement with the analytical results.

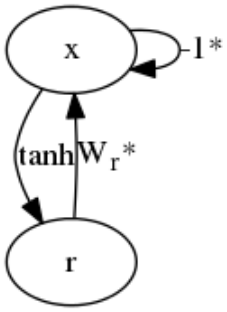


Figure 3: Simple RNN

4 Jacobian and Hessian of a simple RNN

Consider a simplified RNN, as depicted in Figure (3). As in the simple 2D example, we would like to derive analytical expressions for the RNN's kinetic energy $q(\mathbf{x})$, as well as the gradient $\nabla q(\mathbf{x})$ and (approximate) Hessian $\mathbf{H}$.

The dynamics of this particular RNN architecture are as follows:

$$\mathbf{F_i}(\mathbf{x}) = -\mathbf{x_i} + \sum_{\mathbf{k}}^{\mathbf{N}} \mathbf{W^r}_{\mathbf{ik}} \mathbf{r_k} \tag{18}$$

$$\mathbf{r}(\mathbf{x}) = \mathbf{tanh}(\mathbf{x}) \tag{19}$$

The ith row of the Jacobian matrix is then given by

$$\frac{\partial \mathbf{F_i}(\mathbf{x})}{\partial \mathbf{x_j}} = -\delta_{ij} + \mathbf{W^r}_{\mathbf{ij}} \frac{\partial \mathbf{r}(\mathbf{x_j})}{\partial \mathbf{x_j}} \tag{20}$$

Where $\delta_{ij} \iff i = j$. For clarity, the form of the Jacobian is shown in Equation (21).

$$\mathbf{J} = \begin{pmatrix} \square & \square & \square \\ \square & \square & \square \\ \square & \frac{\partial \mathbf{F_i}}{\partial \mathbf{x_j}} & \square \end{pmatrix} = \begin{pmatrix} \square & \square & \square \\ \square & \square & \square \\ \square & W^r_{ij} & \square \end{pmatrix} . * \begin{pmatrix} \square & \square & \square \\ \square & \square & \square \\ \frac{\partial \mathbf{r}}{\partial \mathbf{x_i}} & \frac{\partial \mathbf{r}}{\partial \mathbf{x_j}} & \frac{\partial \mathbf{r}}{\partial \mathbf{x_k}} \end{pmatrix} - \mathbf{I} \tag{21}$$

The nonlinear activation function $r(\mathbf{x_j})$ and its derivative $\frac{\partial \mathbf{r}}{\partial \mathbf{x_k}}$ with respect to $\mathbf{x_j}$ are $tanh(\mathbf{x_j})$ and $1 - tanh(\mathbf{x_j})^{\mathbf{2}}$, respectively.

Once we obtain $\mathbf{J}$, it is straightforward to compute $\nabla q(\mathbf{x})$ and $\mathbf{H}$ via Equations (9) and (13).

5 Echo State Networks (ESNs)

We now consider a more complex RNN architecture: the Echo State Network (ESN). As shown in Figure (4), an ESN receives inputs $\mathbf{u} \in \mathbb{R}^{\mathbf{I}}$, emits outputs $\mathbf{z} \in \mathbb{R}^{\mathbf{O}}$, maintains activations $\mathbf{x} \in \mathbb{R}^{\mathbf{N}}$ and firing rates $\mathbf{r}(\mathbf{x}) \in \mathbb{R}^{\mathbf{N}}$. $\mathbf{r}(\mathbf{x})$ is the element-wise application of a sigmoid nonlinearity (such as the hyperbolic tangent) onto $\mathbf{x}$. A network diagram is depicted in Figure 4.

$W^r \in \mathbb{R}^{N \times N}$ is the matrix of recurrent weights projecting $\mathbf{r}$ back onto $\mathbf{x}$. $\mathbf{W^{FB}} \in \mathbb{R}^{\mathbf{N \times O}}$ are the feedback weights projecting the output back into the activations. $\mathbf{B} \in \mathbb{R}^{\mathbf{N \times I}}$ are the weights for the input layer to the activation layer. Finally, $W^o \in \mathbb{R}^{O \times N}$ are the readout weights from the rates to the output $\mathbf{z}$.

The activation state vector $\mathbf{x}$ obeys the following dynamics:

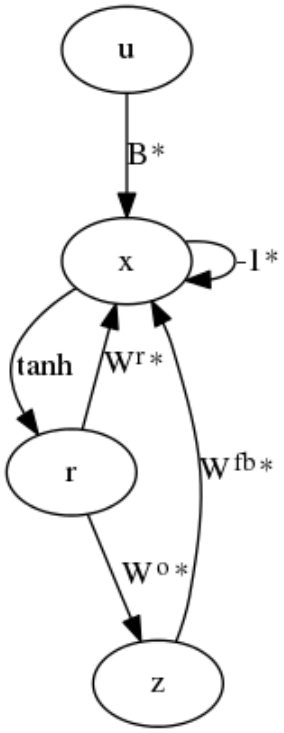


Figure 4: An Echo State Network (ESN) is an RNN that is trained by only varying the output weights, and feeding the output signal back into the network.

$$\mathbf{F}(\mathbf{x}) = -\mathbf{x} + \mathbf{W_r r} + \mathbf{W_{fb} z} + \mathbf{B u} \tag{22}$$
$$\mathbf{z} = \mathbf{W_o r} \tag{23}$$

Our next task is to compute the Jacobian of $q(\mathbf{x})$ for this particular system. Notice that we can take advantage of our previous Jacobian derviation in Equation (21) by substituting Equation (23) into Equation (22) and factoring out the $\mathbf{r}$ term:

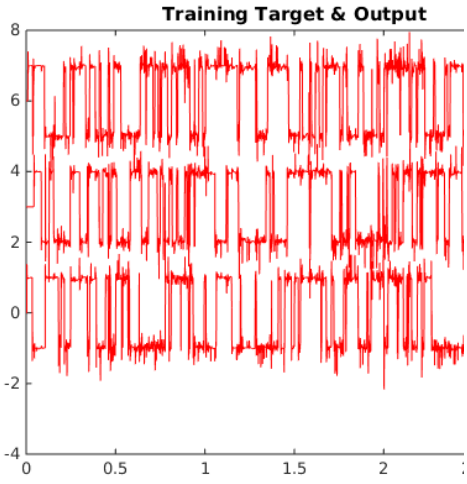
$$\mathbf{F}(\mathbf{x}) = -\mathbf{x} + \mathbf{W_r r} + \mathbf{W_{fb} W_o r} + \mathbf{B u} \tag{24}$$
$$\mathbf{F}(\mathbf{x}) = -\mathbf{x} + (\mathbf{W_r} + \mathbf{W_{fb} W_o})\mathbf{r} + \mathbf{B u} \tag{25}$$
$$= -\mathbf{x} + \mathbf{W^c r} + \mathbf{B u} \tag{26}$$

The form of the Jacobian is computed in exactly the same way as that of the simpler RNN model, except $\mathbf{W^r}$ is replaced with the combined matrix $W^c = (\mathbf{W_r} + \mathbf{W_{fb} W_o})$. The input term $\mathbf{Bu}$ does not depend on $\mathbf{x}$, and so it drops out from the Jacobian.

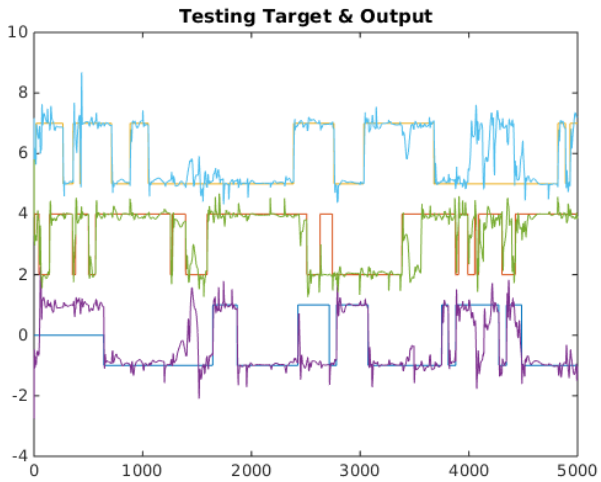
6 Three-Bit Flip-Flop Task

The ESN network consists of an activation state with $N = 1000$ units, three input channels and three corresponding output channels. The three input channels independently hold values of 0, but pulse briefly at random intervals with a value of $+1$ or -1. The ESN is trained to hold each of its outputs equal to the value of the last input pulse on the corresponding channel. Refer to Figure 5.

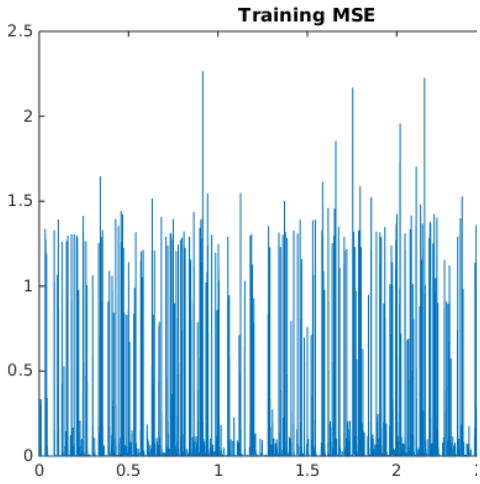
FORCE-learning via Recursive Least Squares was used to train the network to fit synthetically generated training and validation sets. State vectors $\mathbf{x}$ were sampled randomly from test trajectories and used as initial conditions (ICs) for $q(\mathbf{x})$ minimization. In agreement with existing literature,



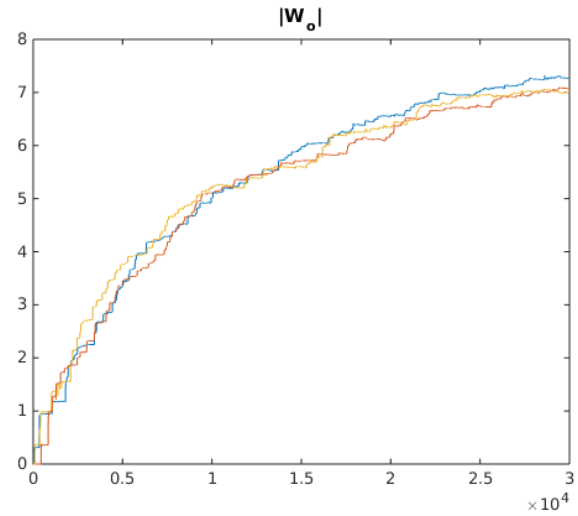
(a) Output of network (red) exactly matches training target (green)



(b) Testing Target and Ouput for 3-bit Flip-Flop



(c) MSE during training is kept small by rapid FORCE learning updates.



(d) Mean weights of $\mathbf{W^O}$ during training.

Figure 5

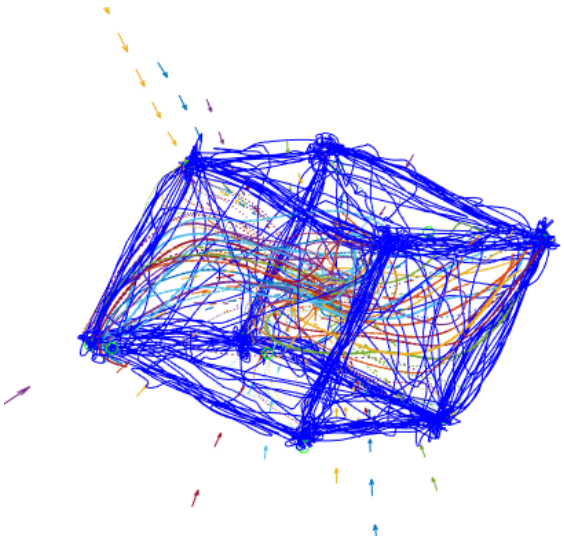


Figure 6: Fixed points, test trajectories, and random sampled ICs projected onto the top three priincipal components. The self-organization of an attracting submanifold illustrates the memory capacity of the network (8 stable states) and the phase portrait shows possible transition paths between these states. See Supplementary Materials online to see an animated roundtable.

$q(\mathbf{x})$ minimization revealed the existence of 9-26 fixed points, depending on the tolerance levels used.

The 1000-dimensional fixed points and test trajectories were then projected to their largest three principal components. Figure (6) shows that the dynamics of the ESN occupy a 3D lattice, with eight attracting fixed points representing the eight distinct memory states (0,0,0), (0,0,1), (0,1,0), ... (1,1,1). Each fixed point corresponds to a memory state that is equidistant from all states one bit-flip away, so the network essentially has spaced out its fixed points in such a way that the "energy" from an impulse is just strong enough to push the state to the attracting set of a different fixed point.

In addition to starting from "naturally-occuring" test states, it is also worthwhile to characterize the dynamics of the system in regions of state space that were not visited during testing. However, what happens to an initial condition that starts inside or outside the cube?

Random output values of $\bar{\mathbf{z}} \in \mathbb{R}^o$ were chosen out of the bounds of stable memory states (for example, $(0.5, 0.5, 0.5)$ or $2.0, 2.0, 0.0$) and used as ICs in Q-optimization. Such trajectories are shown in Figure (6), highlighting the structure of the central fixed point as a saddle.)

6.1 Evolution of Phase Portrait

How does the memory capcity of the ESN gradually emerge over the course of training? The stability and location of fixed points are determined entirely by a rank-three modification (via $\mathbf{W^O}$) to the combined weight matrix, so it is evident that some form of bifurcation is occuring as $\mathbf{W^O}$ is continuously modified via FORCE training.

In this set of experiments, a fixed set of initial states $\{\mathbf{x_i^0}\}$ were chosen, and for different snapshots of $\mathbf{W^O}$ saved during training, $q(\mathbf{x})$ minimization was performed on $\{\mathbf{x_i^0}\}$. As shown in Figure (7), the fixed point at the origin is initially stable, but the phase space is stretched like saltwater taffy into a plane attractor, homoclinic orbits, then a cube attractor.

7 Four-Bit Flip-Flop

We now train the ESN to perform the exact same task, except this time with 4 flip-flop bits rather than three. The expectation is that the fixed points settle on a tesseract in a 4-D manifold, with equidistant, single-bit transitions between neighboring states.

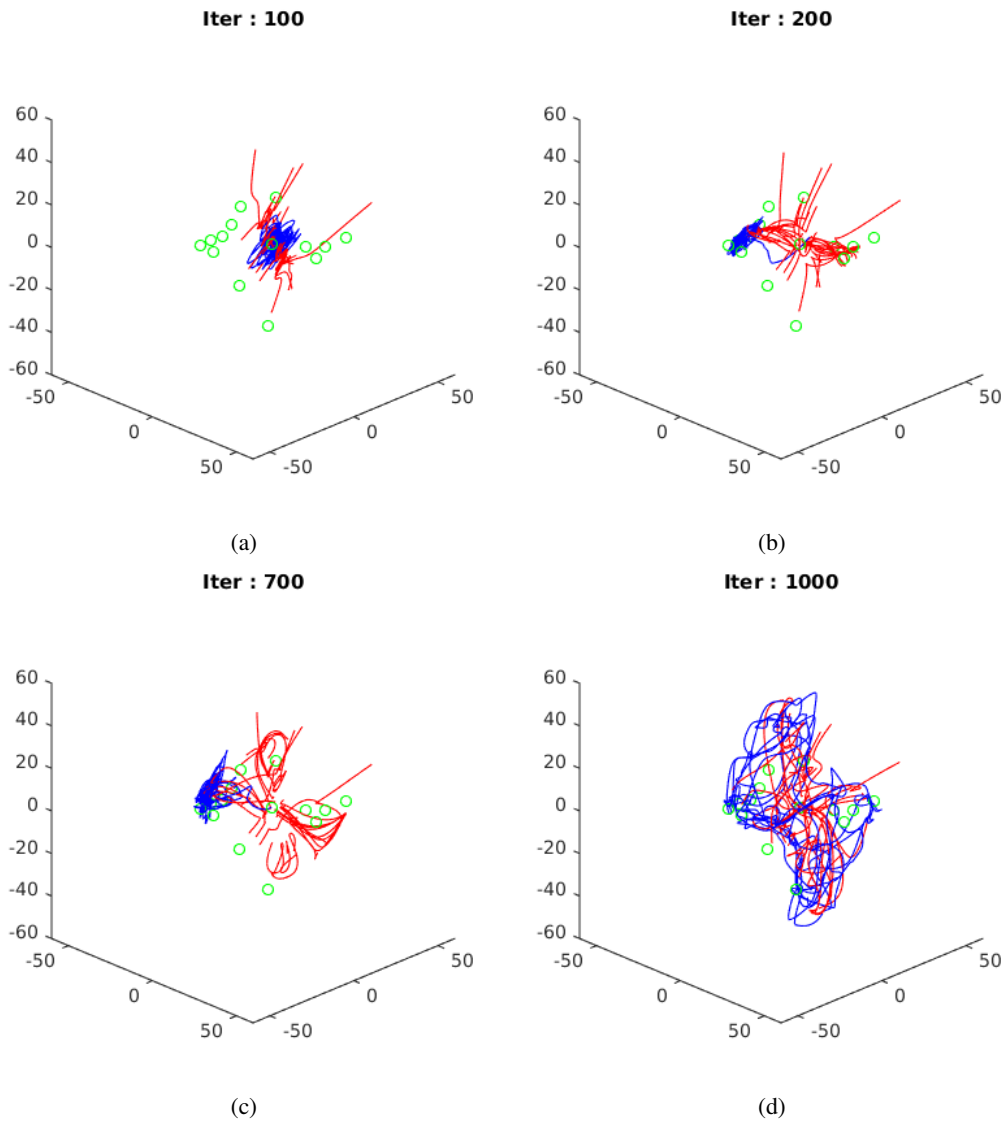
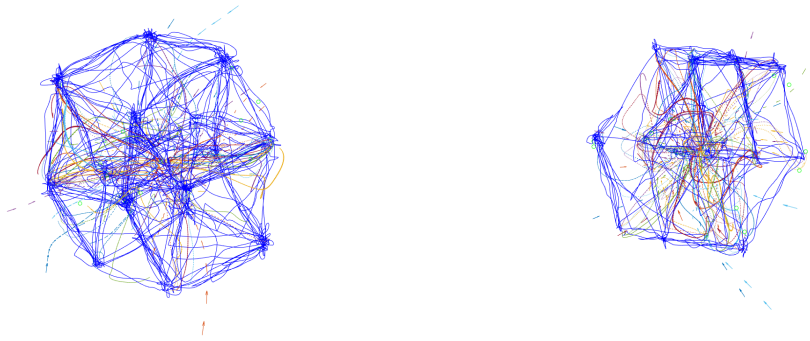


Figure 7: Evolution of phase space over training iterations, with final fixed points overlaid. a) The network is initially at a resting state with a single fixed point at the origin. b) At 200 iterations, the origin is no longer an attracting node. c) At 700 iterations, the principal eigenvalues of the origin vanishes, forming homoclinic orbits (in red). The trajectory (blue) remains fixed on a single memory state. d) The formation of an unstable saddle at the center pushes phase space (red) outwards, gradually bending the test trajectory into a cuboid shape. Full time-lapse animations of the phase space organization can be found online in the Supplementary Materials

.



(a) (b)

Figure 8: (a) Random Initialization, followed by training : tesseract projected edge-first into a hexagonal prism. (b) Transfer Learning from a 3-bit flip-flop : tesseract projected face-first into a cuboid.

The projection of the learned fixed points onto the first 3 principal components is a hexagonal prism (Figure 8a).

7.1 Effect of Transfer Learning on Phase Structure

ANN models often require huge amounts of data to learn patterns, and as such, there is considerable effort in the field directed towards 1) obtaining data, and 2) designing optimal learning policies.

Transfer-learning is a popular technique used to boost performance and reduce the amount of labeled training data required. In transfer learning, the network is initialized with a copy of weights learned on a related task. For instance, suppose that there are not enough images of Black Rhinos to build an accurate "Black Rhino" vs. "not-Black-Rhino" binary classifier from scratch.

Instead, one might train a convolutional neural net to recognize White Rhinos, and then remove the top layer of weights and fine-tune the network against the limited Black Rhino Dataset. Transfer learning saves redundant computation otherwise spent learning "easy" features, by bootstrapping top-level hierarchical representations on top of representations that occur in other similar datasets.

However, it is not clear whether transfer learning merely speeds up convergence to a minimum error, or whether there is a hidden cost to repurposing weights from a different task. The weights themselves are a dynamical system whose trajectories are decided by the supervised learning process.

When a randomly initialized ESN is trained to perform a 4-bit Flip-Flop task, the projection of fixed points lies on a hexagonal prism, such that distances between the 16 stable states are roughly equidistant. However, initializing a 4-bit ESN Flip-Flop with the weights of a 3-bit ESN Flip-Flop (extra dimensions are set to 0). After re-training against the 4-bit Flip-Flop task, the original cuboid manifold is still intact and accomodates 8 new fixed points without morphing into a hexagonal prism (Figure (8)).

8 Robustness to Noise

To assess the stability of memory states, trained 4-bit flip-flop ESNs were subjected to perturbations by adding Gaussian noise distributed along $(\mu = 0, \sigma = \eta)$ to the recurrent weights $\mathbf{W^r}$, for with varying standard deviation levels η.

Networks using either the hyperbolic tangent activation function, or logistic activation function ($r(x) = \frac{1}{1+e^-x}$) performed similarly in response to noise. However, the four-bit flip-flop bootstrapped on top of a three-bit flip-flop was more sensitive to noise, as shown in Figure 9. This may result from memory states in the bootstrapped network not being well-separated enough, so noise becomes more liable to send the ESN into the wrong fixed point.

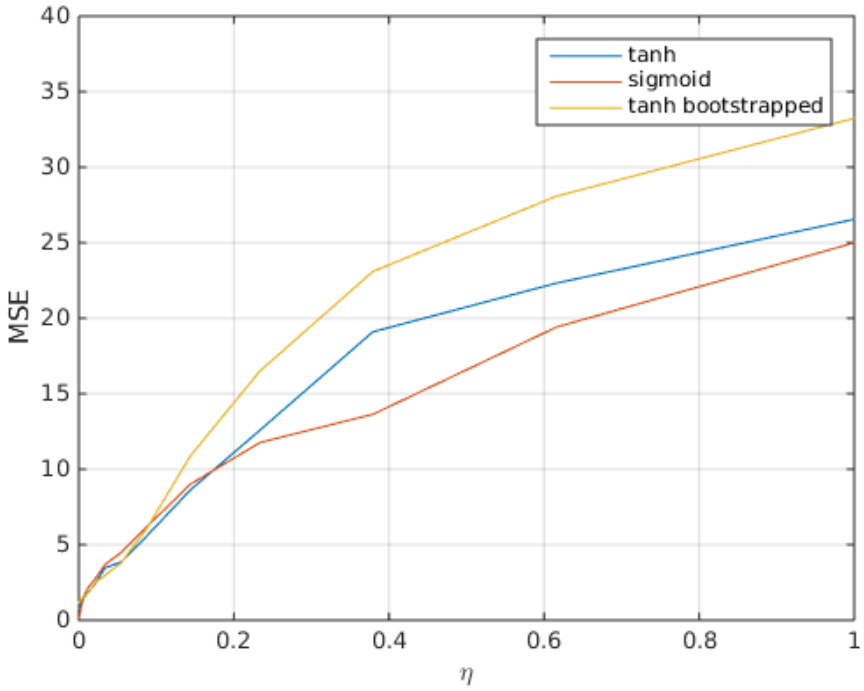


Figure 9: MSE of networks trained to perform the 4-bit flip-flop task, then subjected to random Gaussian noise added to their recurrent weight matrices. Networks with sigmoid nonlinearities are the most robust to perturbations, while a network retrained from a 3-bit flip-flop performed performed the worst.

9 Discussion

Recurrent Neural Networks (RNNs) are suited to problems involving time-dependency of inputs, but historically, they have been hard to train and even harder to understand. This work explores the learning and computation of RNNs from a dynamical systems perspective, in the hopes that understanding the phase space can lead us to better designs for RNN architectures and training algorithms.

An Echo State Network was trained to perform a three-bit flip-flop task. Trained weights were used to derive the kinetic energy function $q(\mathbf{x})$, along with its Jacobian, gradient, and Hessian. This was used in $q(\mathbf{x})$ minimization to identify fixed points, which occupy a cube structure with saddles shaping the transitions from one state to another.

Next, the evolution of the phase portrait was examined as a function of training iterations. Analysis of fixed points at various stages of training progress reveal that the network learns its memories via a series of bifurcations. The single stable node at the origin undergoes a transcritical bifurcation, turning into a saddle. Saddles form along the edges of the cube to guide one memory transition to the next.

Expanding this to a four-bit flip-flop, we see that training produces a tesseract in 4D, which projects to a hexagonal prism in 3D. However, the 3D projection shape is altered and noise sensitivity heightened when the network is pre-trained from the weights of a trained 3-bit ESN. This suggests that initial weight conditions play a huge role in the "geometry of memory".

9.1 Limitations

Given an arbitrary dynamical system, the $q(x)$ minimization procedure cannot automatically distinguish between fixed points and a slow point. There is no clear-cut value for the numerical precision of a "true fixed point"; Sussillo and Barak choose $1e^{-30}$ as the CG tolerance for "fixed points", and $1e^{-27}$ as the tolerance for "slow points", but these values are dependent on the system at in question.

Furthermore, slow and really-slow points are a nice start, but they do not offer the complete picture of the memory capacity of a RNN. RNN "memories" can be encoded by any attracting submanifold in the phase space, not just fixed points. For example, biological neural networks are responsible for generating fixed action patterns, such as egg-rolling behavior in some species of geese. These neural trajectories are attracting, but not invariant (as the goose eventually stops rolling the egg).

9.2 Future Directions

I would like to apply these dynamical systems techniques to futher explore memory capacity and organization in recurrent systems such as Neural Turing Machines (an RNN with a memory matrix) or stigmergic communication in ants (an ant lays down a pheremone and a little while later, another ant picks it up).

10 Methods

This section discusses in further detail the parameters used for network simulation. Source code was written by Eric Jang under the MIT license, and can be downloaded from https://github.com/ericjang/RNN-dynamics

10.1 Network Parameters

Refer to Table 1 for parameters used.

10.2 FORCE-Learning with Recursive Least Squares

In an echo-state architecture, only the readout weights $\mathbf{W^O}$ are modified during training. The key insight of the ESN architecture is that as long as the internal state of the network can be decoded into the correct output, it is irrelevant what the internal state is doing. The recurrent weights $\mathbf{W^r}$ only

Table 1: Network Parameters

Parameter	Value	Description
N	1000	Number of units in activation layer $\mathbf{x}$
g	1.5	Gain of network
alpha	1	Learning rate of the Network
tau	10	Time constant of
δ_{train}	2	Interval between weight updates
Training Duration	30000	Length of each training trajectory
Testing Duration	5000	Length of each testing trajectory

serve the purpose of generating a resevoir of turbulent activity in the network, that can be decoded downstream into a meaningful output.

A problem inherent to training RNNs is that adjusted weights that might reduce errors in the short term might cause later errors to become huge. In typical echo-state network training, this can be compensated for by pretending the network got the output right all the time, and feeding in the target function into the network regardless of the difference between the output and target. The intuition behind this is that if the network output is wrong, one surely does not want to feed wrong output back into the network.

FORCE (first-order reduced and controlled error) is a weight-update scheme for RNNs where output weights are modified on a time scale comprable to network simulation [5]. In FORCE-learning, the feedback is not clamped to the target function, but rather the weights are updated so frequently that all error mangitudes remain small.

At intervals of δ_{train}, matrices $\mathbf{W^O}$ and $P(t)$ are updated via the following:

$$\mathbf{W^o(t)} = \mathbf{W^o(t + \Delta t)} - \mathbf{e_-(t)(P(t)r(t))^\intercal} \tag{27}$$

$$\mathbf{P(t)} = \mathbf{P(t - \Delta t)} - \frac{\mathbf{P(t - \Delta t)r(t)r(t)^\intercal P(t - \Delta t)}}{\mathbf{1 + r^\intercal(t)P(t - \Delta t)r(t)}} \tag{28}$$

$\mathbf{e_-(t)}$ is output error of the current timestep prior to weight modifcation, and $P(t)$ is a running estimate of the inverse correlation matrix between the units in $\mathbf{x}$, which is akin to a correlation-based learning rule.

10.3 Implementation Details

Simulations and analysis were performed on a 4.0 Intel Core i7 workstation. MATLAB's *fminunc* function was used to perfrom conjugate-gradient minimization (with trust-region). $q(\mathbf{x})$ minimization took approximately 3 minutes to find 20 fixed points or ghosts.

Acknowledgments

I thank Dr. John Gemmer for his guidance on my project proposal, and for his communicating the wonder of nonlinear dynamical systems to the APMA1360 class this semester.

References

[1] Kurt Hornik. Approximation capabilities of multilayer feedforward networks. *Neural Networks*, 4(2):251 – 257, 1991.

[2] Awni Y. Hannun, Carl Case, Jared Casper, Bryan C. Catanzaro, Greg Diamos, Erich Elsen, Ryan Prenger, Sanjeev Satheesh, Shubho Sengupta, Adam Coates, and Andrew Y. Ng. Deep speech: Scaling up end-to-end speech recognition. *CoRR*, abs/1412.5567, 2014.

[3] Alex Graves, Greg Wayne, and Ivo Danihelka. Neural turing machines. *CoRR*, abs/1410.5401, 2014.

[4] David Sussillo and Omri Barak. Opening the black box: Low-dimensional dynamics in high-dimensional recurrent neural networks. *Neural Computation*, 25(3):626 – 649, 2014.

[5] David Sussillo and L.F. Abbott. Generating coherent patterns of activity from chaotic neural networks. *Neuron*, 63(4):544 – 557, 2009.