UNFOLDING THE COMPUTATIONAL COMPLEXITY OF RNN THROUGH FIX POINT ANALYSIS

NEURODYNAMICS PROJECT (SS2020)

Rachael Xi Cheng
Master of Cognitive Science
University of Osnabrueck
xicheng@uos.de

Leon Schimd
Master of Cognitive Science
University of Osnabrueck
lschmid@uos.de

Charlotte (Charlie) Lange
Bachelor of Cognitive Science
University of Osnabrueck
chalange@uos.de

November 26, 2021

ABSTRACT

Recurrent Neurla Networks (RNNs) prove to be powerful in processing sequential data, typically time series or language tasks. It is also increasingly used in understanding neurobiological phenomena in real networks of neurons through modeling neuronal connectivity. Yet, the mechanism of the system often remain a black box. Recently, there have been growing interest and efforts in utilizing phase space analysis to reverse engineer how such a nonlinear dynamical system implements the computations. The current project is motivated to replicate and explore locating fixed points and the linearized dynamics around them for understanding the computation mechanism of the RNNs. Three RNN architectures (i.e. Vanilla RNN, LSTM, GRU) were implemented on two tasks - Integration task and FliFlop task. Fix points of different initial conditions were projected to the read out layer. PCA was then performed to portrait the low-dimensional phase space representation. The mechanisms of the trained networks could be analyzed through those steps. We report that while the findings from respective literature exist, it is not consistent and often overshadowed by other effects.

1 Introduction

Efforts to investigate and understand how brain works often search for neural correlates of behavior and explain them through models that link neural activity to behavior. Among the different formal models, there is an increasing use of Recurrent Neural Networks (RNNs) to explain neurobiological phenomena observed in real networks of neurons and as a tool for solving machine learning problems [1]. A hypothesis or an insight can be gained from an appropriately trained RNN model, which reproduces key physiological observations and suggests a new mechanism of input selection and integration [11]

Before diving into how one can use RNNs to study a dynamical system like our brain, there are several prominent features of neuronal networks observed in the brain worth noting [3]. Firstly, it is observed that computation unfolds over time. The brain is capable of holding items in working memory, accumulating evidence towards a decision. Besides, accumulated studies have shown that connections between neurons are rarely unidirectional but more often bi-directional, potentially via more than one synapse. Lastly, neurons perform highly nonlinear operations, which transform inputs to spiking outputs. It results in extremely rich patterns of activity in each neuron, even with a network of small size.

Why are RNNs suitable models for neurobiological activity? In RNN, a given neuron can receive input from any other neurons in the network. Furthermore, activity of neurons in the network is affected by current inputs and

the current state of the network, which contains traces of past inputs. This generates rich intrinsic activity patterns, reminiscent of ongoing activities observed in the brain. [1]. Therefore, RNNs can be understood by being considered as a nonlinear dynamical system and performing phase space analysis on the neurons of the RNNs. Finding the fix points of the RNNs allows to study the phase space around it separately because one can perform linear approximation of the system near those fix points.

Even though no explicit low dimensional structure is built into trained RNNs, using reverse engineering to uncover implicit low-dimensional structure that might have formed implicitly during training has received increasing attention recently [1]. Attractors, saddle points, and other slow areas of phase space are the main defining points of the computation performed by the network and play an instrumental role in finding fixed points of the dynamics. Those key concepts will be illustrated in details in the next sections.

Why is it important to investigate dynamics near a fixed point? For a nonlinear dynamical system, the qualitative behavior of the system varies between different parts of phase space. It is challenging to study systematically. Dynamics near a fixed point are approximately linear. It is therefore easier to systematically locate and analyze, which enables to gain insights into the system's computation mechanism [13].

2 Background: Recurrent Neural Network Architectures

Recurrent Neural Networks have a long history and are one of the key architectures contributing to the success of Artificial Neural Networks and Deep Learning in modeling dynamics in data series, typically time series or language tasks [15] [6]. Generally such a task has a variable sequence length n, resulting in an ordered input sequence of vectors $x_t \in \{x_0, ...x_n\}$ where t denotes the time step. RNN architectures produce two outputs from two inputs: the outputs being (1) cell output o_t and (2) information aggregated for future timesteps as the hidden state h_t with inputs being (1) the input vector x_t and (2) the cell's previous hidden state h_{t-1}. In the following paragraphs, the three types of RNNs used in this project - Vanilla RNN, Long short-term memory (LSTM) and Gated recurrent units (GRUs) - will be reviewed.

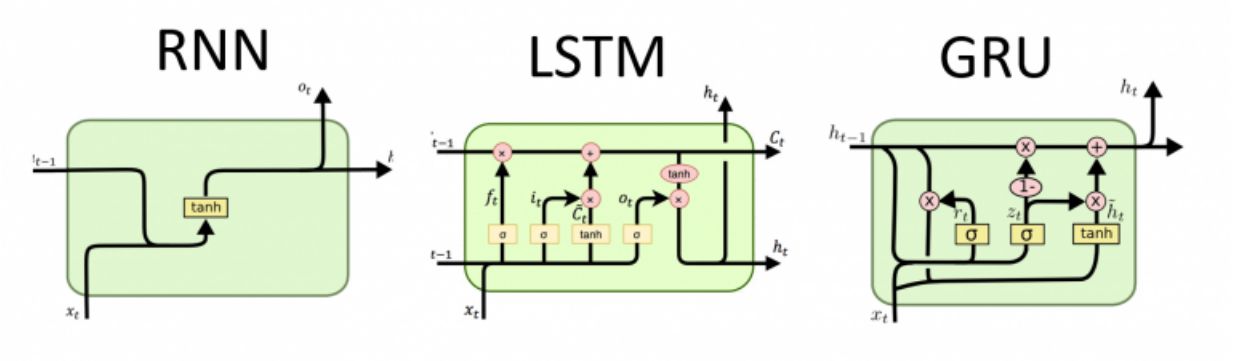


Figure 1: Overview of RNN Architectures [10]. **Left:** A vanilla RNN cell with tanh activation. **Middle:** A common Long Short-Term Memory cell. **Right:** Standard Gated Recurend Unit architecture

2.1 Vanilla RNN

A standard RNN cell as illustrated in figure 1 consists of an internal hidden state h_t, which at each point in time is updated. At each timestep the current input x_t and the previous hidden state h_{t-1} are concatenated and passed through an activation function f - here $tanh$. This activation output is then passed on as the new hidden state of the cell, as well as directly used as the cell output. Hence, a step of a vanilla RNN can be computed according to the following equation:

$$h_t = o_t = f(W_{xh}[h_{t-1}, x_t] + b_{xh})$$

RNNs suffer from the problems of only achieving short-term memory due to the issues of vanishing and exploding gradients during back propagation[2]. These issues are caused by both the effective network depth during backpropagation through time and the repeated derivative w.r.t. the cell activation function[12].

2.2 LSTMs

LSTMs were introduced to overcome these two problems: in a LSTM cell, in addition to the hidden state as in RNN cell, a separate cell state is created to serve as long term memory. It is therefore able to pass information over longer input/time series. Manipulation of this cell state is regulated through the so-called gates, which are specifically designed to eliminate issues like exploding or vanishing gradients during backpropagation through time. Each gate contains a sigmoid activation layer, scaling the input to the range $[0, 1]$, which by pointswise multiplication / Hadamard multiplication in turn is used to model the effect size on each cell entry.

LSTMs consist of four important parts - forget gate, input gate, cell state update, and output gate:

- **forget gate**: It is used to remove information from the cell state, either due to it becoming irrelevant or even counterproductive to the performance. The forget gate f_t is computed as a fully connected layer from the concatenation of previous hidden state activation and input with a sigmoid activation. As described above this results in activations in the range $[0, 1$ which are then used in pointwise/Hadamard multiplication with the previous cell state effectively removing parts of it by multiplication with values close to zero.

$$f_t = \sigma(W_f[h_{t-1}, x_t] + b_f)$$

- **input gate**: It is used to store relevant information from the current input in the cell state. Two fully connected layers are computed from the the concatenation of previous hidden state h_{t-1} and the input x_t: The first makes use of $tanh$ activation function, resulting in activations in the range $[-1, 1]$ granting the ability to manipulate cell state in both positive and negative directions. Then again the second layer makes use of sigmoid activation and by pointwise/Hadamard multiplication with the first allows decisions on which parts of it to actually apply. This results in the input gate vector:

$$i_t = \sigma(W_i[h_{t_1}, x_t] + b_i) * tanh(W_c * [h_{t-1}, x_t] + b_c)$$

  It is bound to the range $[-1, 1]$ and is simply added to the cell state in the subsequent cell state update

- **cell state update**: The cell state update applies the forget and input gates to the previous cell state c_{t-1} to produce the current cell state c_t. It first applies pointwise multiplication with the forget gate and then pointwise addition with input gate, which have both been described in detail above.:

$$c_t = f_t \cdot c_{t-1} + i_t$$

  This cell state update is resilient against exploding and vanishing gradients and can achieve long term memory over several hundred timesteps[4].

- **output gate**: It computes hidden state for the timestep, whis is also used as the respective cell output for the timestep. A fully connected layer with sigmoid activation is applied to the concatenation of previous hidden state and input. Again this is used to select elements by pointwise multiplication, here with the (updated) cell state passed through a tanh function. Effectively this results in the output being a scaled and pointwise selected version of the cell state.

$$o_t = \sigma(W_o[c_{t-1}, x_t] + b_o) \cdot tanh(c_t)$$
$$h_t = o_t$$

As pointed out before LSTMs mostly solve the problem of vanishing and exploding gradients[7] with backpropagation through time, by having their cell state only accessible via purpose built gates. This enables them to solve long term memory tasks and made them the state of the art architecture for sequence and time-series tasks, only recently being challenged by attention based architectures. [14]

2.3 GRUs

A much more recent offshoot of the LSTM architecture Gated Recurrent Units (GRUs) offer a simplified version of the LSTM mechanism, whose effectiveness while heavily discussed at least challenges LSTMs on some tasks [8]. Mostly the difference to the LSTM can be summarized by (1) merging cell state and hidden state and (2) merging input gate and forget gate.

- **Reset Gate**: the reset gate makes a new hidden state proposal $\tilde{h}_t$ as it decides which parts of the previous hidden state h_{t-1} to combine with the current input x_t

$$r_t = \sigma(W_r[h_{t-1}, x_t] + b_r)$$
$$\tilde{h_t} = tanh(W_h[h_{t-1}, x_t] + b_h) \cdot r_t$$

- **Update Gate**: the update gate decides how much of the previous hidden state is kept and what to keep from the hidden state proposal of the reset gate. These two parts are then added to become the new final hidden state.

$$z_t = \sigma(W_z[h_{t-x}, x_t] + b_z)$$
$$h_t = z_t \cdot h_{t-1} + (1 - z_t) \cdot \tilde{h_t}$$

While their performance gain as compared to LSTMs is heavily contested [8], we consider GRUs for our experiments as they are both mentioned in the respective literature and reports our project is based on. Moreover, they are also attractive to our experiments by outperforming RNNs with not needing a second hidden state (the cell state in LSTMs), which is troublesome for our fix point analysis.

3 Background: State Space Analysis

The dynamics of neural network models can be represented by a set of first order differential equations:

$$\frac{d}{dt}\boldsymbol{X}(t) = \boldsymbol{F}(\boldsymbol{X}(t)$$

.

Here the nonlinear function $\boldsymbol{F}$ operates on elements of the state vector $\boldsymbol{X}(t)$. $\boldsymbol{X}(t)$ is the state of the system at particular time t. $\boldsymbol{F}(\boldsymbol{X}(t))$ is a vector field in an N-dimensional state space and does not depend explicitly on time t.

Phase space of the dynamic system describes the global characteristics of the motion, instead of the detailed aspects of analytic or numeric solutions of the equation. This can be used to better visualize the motion of the states in time.

3.1 Fix Points and Slow Points

In state space analysis, fix points play an important role. As the behaviour of a nonlinear system varies greatly among its phase space and is therefore difficult to understand, different parts of the phase space need to be studied separately. When analyzing dynamical systems, it is convenient to first concentrate on its equilibria (in the following noted as fix points). They refer to the points in the state space of the system where the state variables do not change. Thus, given a system characterized by the equation $F(V)$ (where V is a function of some variable(s)), we look for the values where the following conditions hold:

$$F(V) = 0$$
$$\dot{V} = 0$$

If the initial state is exactly at such a fix point, the value of the state variable will not change over time. However, if the initial state is near the value of the fix point, different behaviours can be observed. The trajectory will over time diverge from the nearby fix point in a specific dimension or converge to the fixed point or a cycle.

Therefore, a fix point can be characterized by the behaviour of the system in it's proximity. A fix point can thus be stable (trajectory near it converge to this point) in all dimensions, making it an attractor. A fix point unstable in all dimensions is called a repellor. Also, fix points can be bistable, thus unstable in only some dimensions.

These fix points are useful when it comes to explaining the behaviour of a dynamic system and in particular the interaction of different attractors. For example, such a bistable point could be a saddle point with only one unstable mode may use all its stable modes as a funnel and use the unstable mode to push the system to different attractors depending on the direction of its unstable mode. Such an example is also called a line attractor.

However, true fix points might not be the only regions of interest when it comes to complex dynamical systems. Susillo et al. [13] suggest to also look at the "slow points" - regions in the state space where the value of $F(V)$ is not exactly 0 but very small with a lower bound of 0. Hence these points are not perfect but only near perfect equilibria.

Not only are fix and slow points of a dynamical nonlinear system in itself interesting, they also are good candidates for local linear approximation of the system. This is desirable, because little is known about high dimensional complex nonlinear dynamical system in contrast to linear ones, thus exploring the linear dynamics around the fix points can help greatly in interpreting and explaining the dynamics of the model.

That fix and slow points are reasonable candidates for such a linear approximation can be illustrated with the help of the Taylor series expansion. Given a nonlinear system described by the motion equation $\boldsymbol{F}$ and a N-dimensional state vector x let us consider a first-order differential equation system:

$$\dot{x} = F(x)$$

We aim to find values $\overset{*}{x}$ around which linear approximation would be valid. Using the Taylor expansion around $\overset{*}{x}$ we get

$$F(\overset{*}{x} + \delta x) = F(\overset{*}{x}) + F'(\overset{*}{x})\delta x + \frac{1}{2}\delta x F''(\overset{*}{x})\delta x + \textit{higher order terms}$$

Since we are looking for fix and slow points, $\overset{*}{x}$ only have very small perturbations around them, hence $\delta x \equiv x - \overset{*}{x}$ and since we are interested in a linear approximation, we want the first order derivative of the right hand side of the Taylor series expansion to dominate all the other terms. These conditions give rise to the following inequalities:

$$|F'(\overset{*}{x})\delta x| > |F(\overset{*}{(x)}|$$

$$|F'(\overset{*}{x})\delta x| > |\frac{1}{2}\delta x F''(\overset{*}{x})\delta x|$$

As for true fix points it holds that $F(\overset{*}{x} = 0$ they are evidently good candidates for linearization since then δx is bounded from below by 0. Moreover, slow points can be considered with a positive nonzero but very small value of $F(\overset{*}{x}$.

3.2 Speed Optimization

Following this line of thought, points (fixed or slow) that are valid for linearization can be found by minimizing $|F(x)|$. As proposed in [13], it can be achieved by designing an auxiliary function $q(x)$, which can be considered as being equivalent to the speed or the kinetic energy of the system:

$$q(x) = \frac{1}{2}|F(x)|^2$$

$q(x)$ is introduced because it can explicitly show when the function reaches a minimum.

Since we want to find points $\overset{*}{x}$ of the system where the dynamics remain unchanged, we would like to minimize the difference between points in the system's state space and the output of our system at these points. Hence, we can define our fix point loss as the following:

$$(x - F(x))^2)$$

The respective optimizationc an be summarized as follows then:

Data: C_c fix point candidates
Result: C_f final fix points
initialize loss function f, optimizer;
load model;
initialize constant $\overset{*}{x}$;
while *loss* < *tolerance* **do**
  $h \leftarrow$ model($\overset{*}{x}$);
  loss $\leftarrow f(C_c, h)$;
  bachpropagate loss, update C_c;
end
$\mathrm{C}_f \leftarrow C_c$

Algorithm 1: Finding Fix Points

4 Experiments and Results

In our experiments we test the three RNN architectures described above on two different tasks, which in turn can be varied in (1) task size and (2) network size. In this section we first explain these experiments in more detail and then report on experiment results and discuss them in detail.

4.1 Experiment Tasks

The two kinds of tasks we run experiments on are either taken from the literature this project is based on [13] or motivated by a related project & blog entry on github[**google_research**].

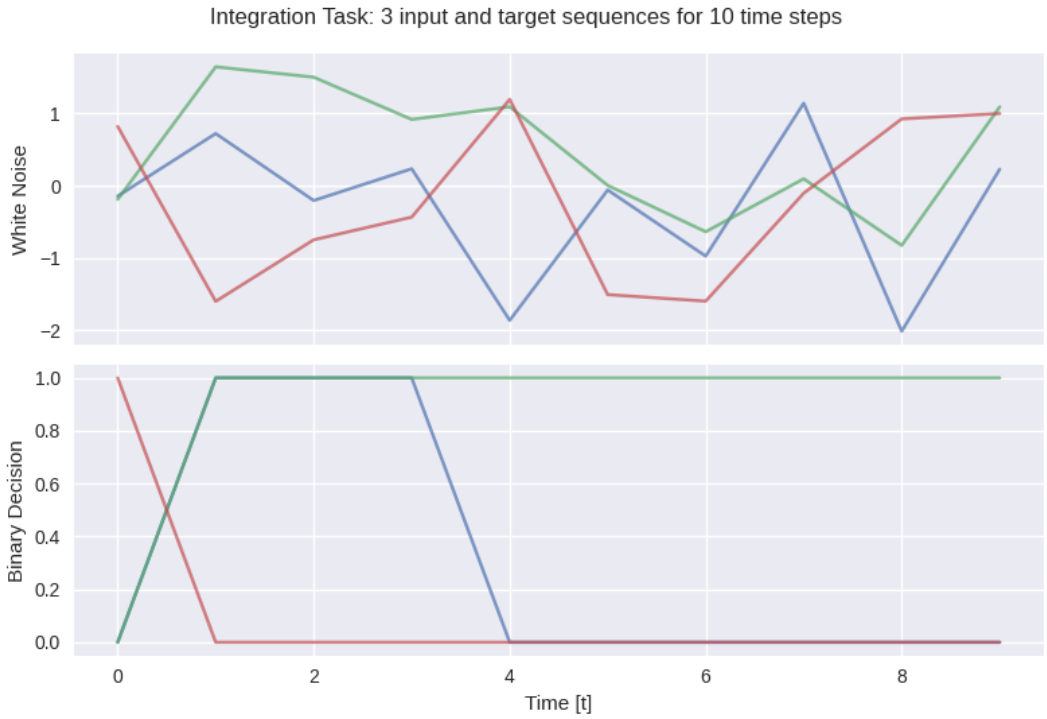


Figure 2: Integrator Task: **Top:** White noise input sample from a normal distribution. **Bottom:** Binary target over time. 0 means negative and 1 positive integral.

4.1.1 Integration Task

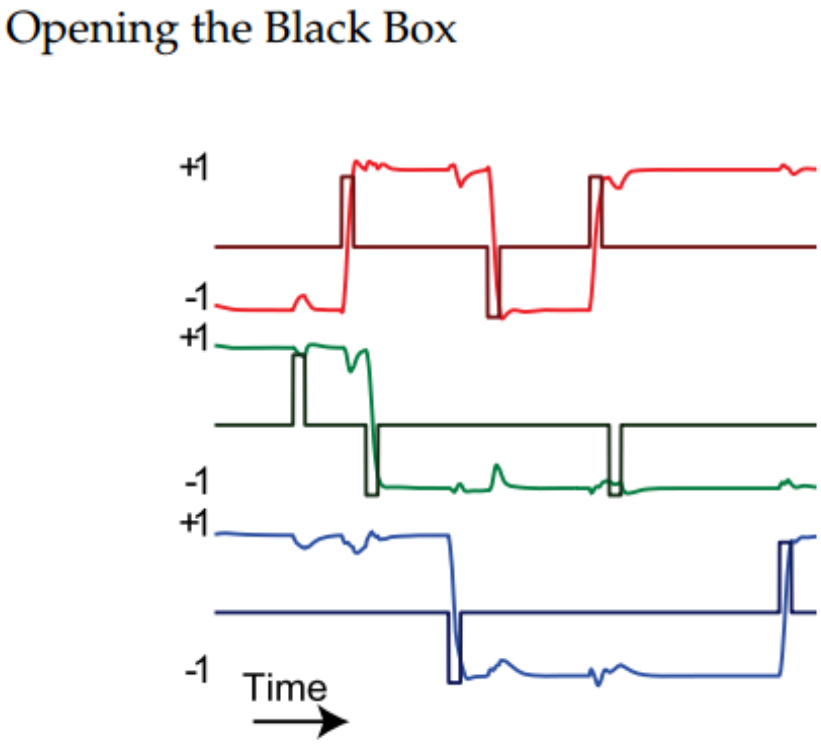
To reproduce the results of Sussilo et al. [13], we chose the example of the "Integration Task" which is similarily used in the released source code of the same paper. As illustrated in figure **??** the input consists of white noise channels over time, n_t denotes the noise signal in a specific channel at timestep t. The target is a binary decision on whether the integral of the white noise up to that timestep is positive or negative. Such an integration task can be posed on multiple channels, where each channel has it's own noise input channel, and own target channel, but all channels are processed through one singular network. For further investigation we implement an exponentially discounted sum version of the task, where a discount factor $\gamma \in [0, 1[$ is used to produce the discounted sum s_T up to timestep T of $s_T = \sum_{t=0}^{T} \gamma^T n_t$. Again also in this discounted version the target is a decision on whether this discounted sum is lower or higher than zero. This discounted version of the task is interesting for studying input dependent fixpoints later.

4.1.2 FlipFlop Task

We consider a second task, named Flipflop task, from the github repository that accompanied the literature this project is based on. Effectively it is based on long term memorization of events in single channels - a procedure we streamline further for our implementation. Our implementation again makes use of multiple channels, which are mutually independent of each other. Each channel has a default input value of one. With a probability p_c a channel is *flipped* by randomly showing either +1 or -1 as input. The target is to predict for each channel the last non-zero input. We default our experiments to having the probability of a flip p_c be 0.1. This task again is enticing because in the way we state it, a zero input should not generate any change in hidden state and we expect to find clustered fixpoints for such a 0 input.

4.2 Training Details

We trained the three above mentioned RNN architectures, namely a vanilla RNN cell, LSTM and a GRU, on both tasks. We consider both tasks with 3 channels, and additionally the integration task with only one channel. All architectures were using the same number of units in their recurrent cell (100) and were trained equally long. We use (channelwise binary) cross entropy loss and an ADAM optimizer [9] with default hyperparameters for optimization, weights are initialized randomly using Glorot initialization[5]. During training almost no noticeable differences could be observed



Input signal and target for FlipFlop task with 3 channels, as taken from [13]

except that as could be expected the vanilla RNN cell took slightly longer to learn the task. All architectures learn the task very well with accuracies consistently above 95% on test data.

4.3 Fix Point Extraction

We extract slow points and fixed points by closely following the procedure from[13]. We again use gradient decent via backpropagation, but contrary to training the RNNs we fix the network weights. Instead we model singular steps through the network with a fixed input x_f. The second necessary cell input is h_{t-1}, the previous hidden state, which we use as the trainable parameter here. We minimize the mean square error (MSE) between this previous hidden state and the one produced by the network (h_t) when it is presented with it and the fixed input:

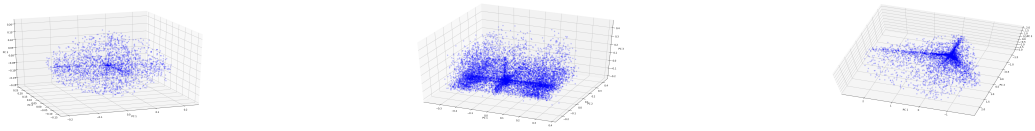
$$argmin_{h_{t-1}}(h_{t-1} - h_t)^2$$

This optimization yields such hidden values h_{t-1}, such that with the given input their speed, i.e. their change as compared to the hidden state produced from them is minimized. This effectively yields slow points and fix points, the batch average speed in our training process consistently drops below 0.01, typically much lower. As initial values we use candidate values generated from sampling actual hidden states from runs on the training data set. This ensures that the initial candidates are from the distribution of hidden states actually encountered by the network and thereby keeps the generated slow points and fix points close to those values. This is important to ensure they are actually meaningful for describing the RNN dynamics.

4.4 Results and Analysis

We analyze the resulting fix points by plotting them in 3D space using Principal Component Analysis (PCA) for dimension reduction. As can be seen in Figure 3 the underlying structure is stable for all three network architectures.

Given this strong similarity we concentrate most of our analysis of the different experiments on their GRU version here.



(a) PCA of fixpoints of GRU trained on 3-channel Integration task

(b) PCA of fixpoints of LSTM trained on 3-channel Integration task

(c) PCA of fixpoints of RNN trained on 3-channel Integration task

Figure 3: PCA plot of fixpoints of all three different network architectures trained on the same task. Explained variance of first 3 dimensions is similar across all of them and above 90% consistently. The underlying structure of three perpendicular axes which the fix points cluster around is present in all three of them. GRU exhibits the cleanest structure.

4.4.1 Single Channel Integration Task

The single Channel Integration Task runs the Integration task described in 4.1.1. with only a single channel and a discount factor of 1.0 .

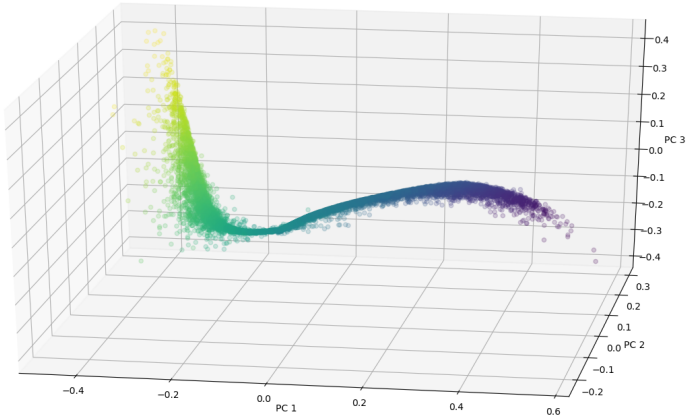


Figure 4: Single channel Integration task, PCA of fix points colored by respective read-out layer value mapping

Results from the single channel integration task show strong similarities to [13]. With over 90% of variance explained by the first principal component (PC1) results are very close to what had to be expected: The most informative axis PC1 simply maps the current state on a preference nearly linearily to either of the two classes, as can be seen in the color read out. While the subsequent axes PC2 and PC3 have much lower explanatory power - being responsible for much less of the explained variance than PC1 - they still exhibit a similar structure to Sussilo et al. [13]. The only really decicive readouts happen in the violet and yellow colored regions (Figure 4), where we can confirm strong curvature [13].

By using a simple extension of methods we can further clarify these results: We further differ fixpoints by their respective responsible input. Specifically we realize that Sussillo et al. only create fixpoints optimized on all zero inputs $x_t = [0, ..., 0]$. We conclude that these fixpoints only show such points, where an input of zero should not change the state to much. We postulate that further insight into the dynamics of RNN architectures can be gained by analyzing input-dependent fix points and introduce further fix points optimized on inputs of one and negative one: $x_t = [1, ..., 1]$ and $x_t = [-1, ..., -1]$.

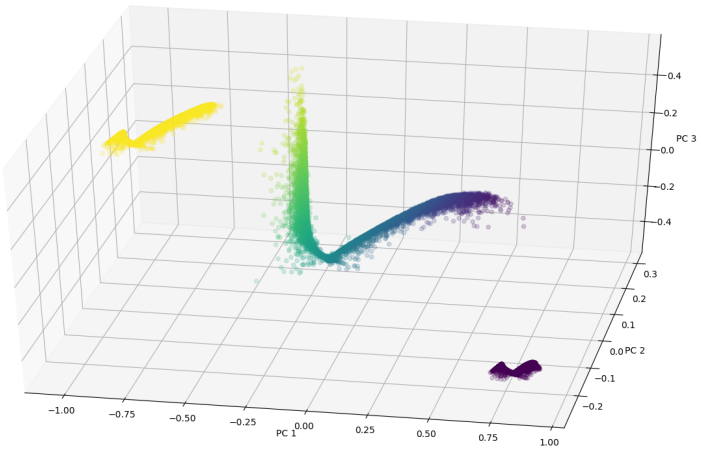


Figure 5: Single channel Integration task, PCA of fix points colored by respective read-out layer value mapping, including input dependent fix points for inputs of 1, 0, and -1 .

Figure 5 in our opinion succeeds at further clarifying these dynamics: As already postulated by Sussillo et al. [13], this further supports the idea that trajectories along fixpoints in a task like the integration task at hand encode how sure the network is of it's classification. We can see fix points for fixed input of one clearly clustering and being classified by the network as outputs depicting positive sums. As they also naturally extend the fixpoint structure trained on zero inputs at the tail that is already classified as positive sums, we conclude that this area indeed is used to encode hidden states where the sum of inputs already is skewed so strongly towards a positive sum, that further input does not, or only marginally change the hidden state anymore. The same point can be made for inputs of -1 on the opposite side of the structure. We conclude that this supports the hypothesis, that the hidden state rather encodes the sureness about classification, i.e. low probability of this classification changing given future input, rather than any reasonable prediction of the actual sum.

4.5 Three Channel Integration Task

We further push the Integration Task experiment to it's three channel version. Figure 3 and Figure 6 clearly explain the structure for fixed zero input fix points in this task. Contrary to similar experiments in literature and accompanying blog and github entries [13] we can not report a cube like structure. Rather we find fix points being clustered around three perpendicular axes. We assume these axes coorespond to the three channels used in the task. While the cube structure from literature has a clear explanation (each corner representing one combination of the three binary channel targets), the structure we observe fails at producing such an explanation in our opinion. While it seems intuitive that each axis is responsible for one of the three channels, the places of high fix point structure fail at explaining any reasonable three dimensional target in our opinion.

4.6 Three Channel FlipFlop Task

Finally we consider a three channel Flipflop Task experiment. As can be seen in Figure 7, we again fail to reproduce the cube like structure from literature and respective blog and github pages [13]. Instead we again get a very similar structure to the three channel integration task of fix points clustered around three perpendicular axes. We again assume that these are an internal representation of the three channels. Figure 8 revisits the idea of input dependent fix points. We can clearly see the cluster of fix points trained on zero inputs in the middle, with the clusters on negative one and positive one inputs behaving mirrored on PC1 (which holds the most information), and close to equal in PC2 and PC3. Taking a closer look at the input dependent fix points in Figure 9 grants two additional interesting insights: Not only do these fixpoints have an extremely similar structure to the one of zero inputs internally (right image), featuring the same

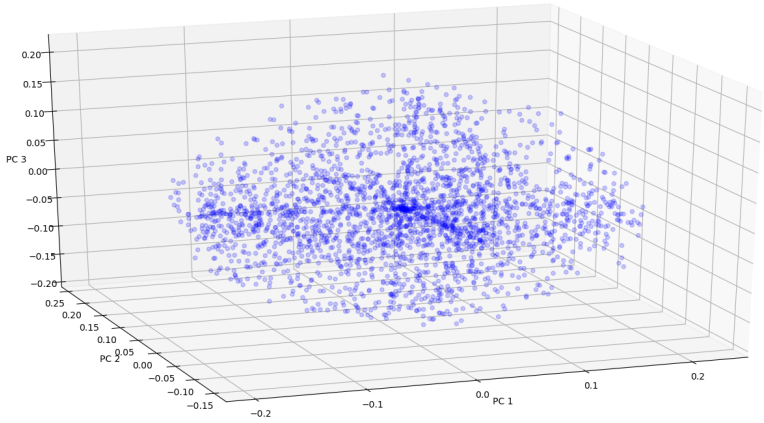


Figure 6: Three channel Integration task, PCA of fix points only trained on zero input

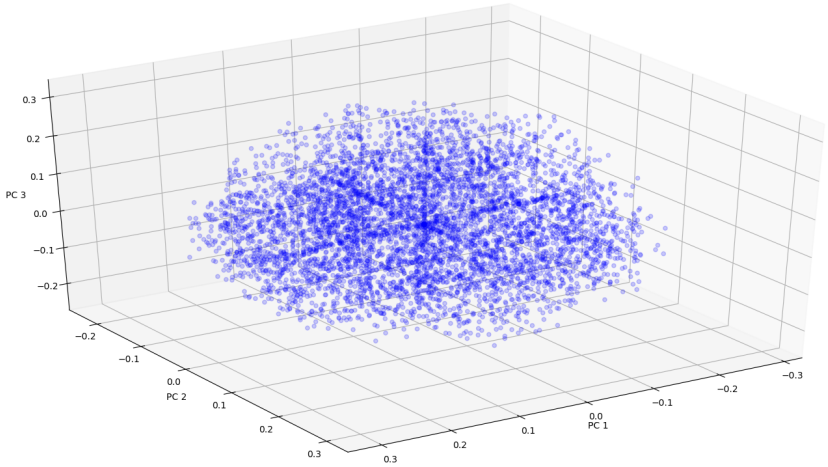


Figure 7: Three channel FlipFlop task, PCA of fix points only trained on zero input

clusters along three perpendicular axes, but repeating two of those as parallels. We can also see (left image) that they seem to be oriented in the same way in the hidden state space.

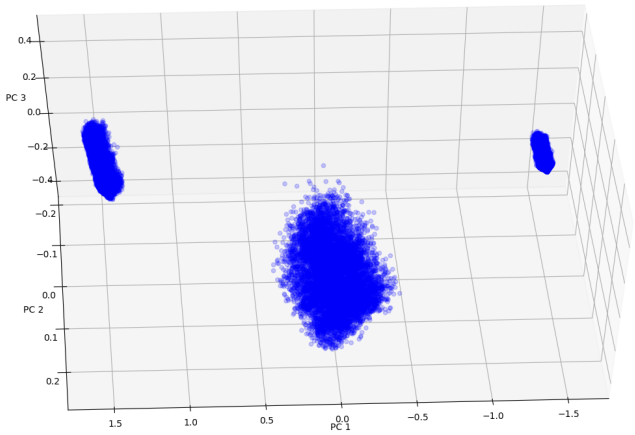


Figure 8: Three channel FlipFlop task, PCA of fix points only trained on zero input

(a) Three channel FlipFlop task, PCA of fix points only trained on 1 and -1 inputs

(b) Three channel FlipFlop task, PCA of fix points only trained on 1 input

Figure 9: A closer look at input dependent fix points in three channel FlilFlop task, PCA explains more than 90% of variance.

5 Discussion

The results we achieve to not share the same structure to the underlying literature [13]. While we indeed encountered the cube structure described there while running multiple experiments to test our setup, we can not consistently report them as results and typically end up with the fix point structure described in the section above.

Along with this criticism also produce other evidence, that the interpretation of fix points in literature might be partially faulty. Figure 10 shows such an example, which is based on the Single Channel Integration Task. It clearly shows how fix points do not have any clear impact or interpretation regarding the actual trajectories of hidden states.

While these findings could be caused by faults in implementing the experiments on our side, we believe our experiments give good indication, that fix point analysis as a tool for interpreting RNNs should be further reviewed. We propose that specifically input dependent fix points, as well as fix points trained on the actual input distribution of the respective tasks might be important in this task.

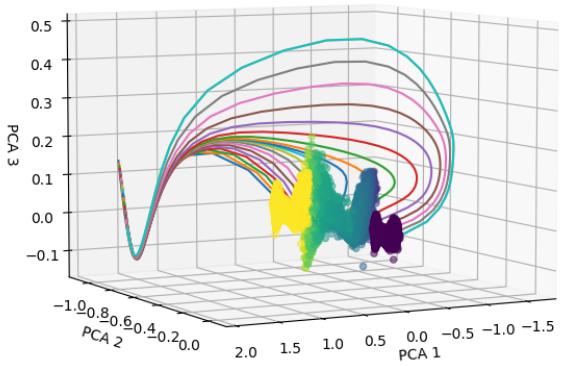


Figure 10: PCA of Fix Points in Single Channel Integration Task, additionally exemplary trajetories of actual hidden states during selected test runs are depicted

References

[1] Omri Barak. "Recurrent neural networks as versatile tools of neuroscience research". In: *Current opinion in neurobiology* 46 (2017), pp. 1–6.

[2] Yoshua Bengio, Paolo Frasconi, and Patrice Simard. "The problem of learning long-term dependencies in recurrent networks". In: *IEEE international conference on neural networks*. IEEE. 1993, pp. 1183–1188.

[3] Sebastian Bitzer and Stefan J Kiebel. "Recognizing recurrent neural networks (rRNN): Bayesian inference for recurrent neural networks". In: *Biological cybernetics* 106.4-5 (2012), pp. 201–217.

[4] Felix A Gers, Nicol N Schraudolph, and Jürgen Schmidhuber. "Learning precise timing with LSTM recurrent networks". In: *Journal of machine learning research* 3.Aug (2002), pp. 115–143.

[5] Xavier Glorot and Yoshua Bengio. "Understanding the difficulty of training deep feedforward neural networks". In: *Proceedings of the thirteenth international conference on artificial intelligence and statistics*. 2010, pp. 249–256.

[6] Alex Graves, Abdel-rahman Mohamed, and Geoffrey Hinton. "Speech recognition with deep recurrent neural networks". In: *2013 IEEE international conference on acoustics, speech and signal processing*. IEEE. 2013, pp. 6645–6649.

[7] Sepp Hochreiter and Jürgen Schmidhuber. "Long short-term memory". In: *Neural computation* 9.8 (1997), pp. 1735–1780.

[8] Łukasz Kaiser and Ilya Sutskever. "Neural gpus learn algorithms". In: *arXiv preprint arXiv:1511.08228* (2015).

[9] Diederik P Kingma and Jimmy Ba. "Adam: A method for stochastic optimization". In: *arXiv preprint arXiv:1412.6980* (2014).

[10] dProgrammer lopez. *RNN, LSTM GRU*. URL: `http://dprogrammer.org/rnn-lstm-gru`.

[11] Valerio Mante et al. "Context-dependent computation by recurrent dynamics in prefrontal cortex". In: *nature* 503.7474 (2013), pp. 78–84.

[12] Razvan Pascanu, Tomas Mikolov, and Yoshua Bengio. "On the difficulty of training recurrent neural networks". In: *International conference on machine learning*. 2013, pp. 1310–1318.

[13] David Sussillo and Omri Barak. "Opening the black box: low-dimensional dynamics in high-dimensional recurrent neural networks". In: *Neural computation* 25.3 (2013), pp. 626–649.

[14] Ashish Vaswani et al. "Attention is all you need". In: *Advances in neural information processing systems*. 2017, pp. 5998–6008.

[15] Yong Yu et al. "A review of recurrent neural networks: LSTM cells and network architectures". In: *Neural computation* 31.7 (2019), pp. 1235–1270.