VSA Next Generation Reservoir Computing

Ross Gayler

2023-04-01

Table of contents

0.1 Objectives

The aim of this project is to develop good VSA implementations of Next Generation Reservoir Computing (Gauthier et al. 2021). The sought outcomes are:

- A good, practical multivariate dynamical system predictor;
- Further theoretical development of VSA through exposure of current understanding to new practical problems. (I view theoretical development as being about finding productive interpretations/extensions of the abstract basis of VSA.)

0.2 Statistical feature construction

0.2.1 What does Reservoir Computing do?

Reservoir computing uses a recurrently updated reservoir to create a representation vector from a sequence of input values. The reservoir value is transformed to predictions by a simple linear readout transformation, for example see (Gauthier et al. 2021, fig. 1).

0.2.2 Interpretation as statistical feature construction

The linear readout is equivalent to the application of a standard statistical regression model. A standard univariate regression model is effectively the calculation of the dot product of the vector of regression coefficients and the vector of predictor values. A multivariate regression is the concatenation of multiple univariate regressions (so the vector of regression coefficients becomes a matrix of regression coefficients). Under this interpretation, the reservoir is constructing the vector of predictor values to be input to the regression model.

0.2.3 VSA representation of predictors

In standard statistics/ML the representation of predictor values is localist. However, in VSA the natural form of representation is distributed key-value pairs, with the key corresponding to the identity of the predictor and the value corresponding to the value of the predictor (Kanerva 1997). I *think* that the distributed representations can be viewed as rotations of a localist representation (Qiu, n.d., sec. 6.1).

It's also worth noting that in standard statistics/ML the value of a predictor is represented as the scalar value of the predictor variable. This can be directly implemented by representing the value of the predictor variable by the magnitude of the hypervector (e.g. Kleyko, Frady, et al. (2022), sec IV.A.2). Note that this represents the predictor as a key rather than a key-value pair. An alternative is to represent the value of a predictor by the direction of a hypervector (e.g. by Fractional Power Encoding).

0.2.4 Theory of sequence indexing

The theory of sequence indexing and working memory in recurrent neural networks (Frady, Kleyko, and Sommer 2018) appears to be very relevant to this project, but I haven't read it in detail yet. On the basis of a quick skim this project appears to differ by stressing the algebraic interpretation of the action of the reservoir update. In this project we interpret the reservoir update as algebraic VSA operations.

0.2.5 Integer Echo State Network

We'll start by applying this lens (algebraic interpretation of the reservoir update) to the Integer Echo State Network. The reservoir update equation (Kleyko, Frady, et al. 2022, eqn. 5) is:

$$\mathbf{x}(n) = f_\kappa(Sh(\mathbf{x}(n-1), 1) + \mathbf{u}^{\mathrm{HD}}(n) + \mathbf{y}^{\mathrm{HD}}(n-1))$$

where $\mathbf{x}(n)$ is the value of the hypervector representing the state of the reservoir at time n; $f_\kappa()$ is the clipping function applied to the reservoir hypervector; $Sh(\mathbf{x}(n-1), 1)$ is the application of a cyclic shift (by one element) permutation to the value of the reservoir hypervector at the previous time step $(n-1)$; $\mathbf{u}^{HD}(n)$ is the hypervector encoding of the input vector at time n; $\mathbf{y}^{HD}(n-1))$ is the is the hypervector encoding of the output vector at the previous time step $(n-1)$; and $+$ is hypervector addition (bundling).

Simplify the equation and notation a little for ease of exposition:

- Drop the clipping function $f_\kappa()$ (equivalent to setting the clipping threshold κ to a high value).
- Replace the cyclic shift function $Sh()$ with a generic, fixed permutation function $\mathrm{P}()$.
- Drop the HD superscripts from the input and output hypervectors by interpreting $\mathbf{u}(n)$ and $\mathbf{y}(n-1))$ as the hypervector encodings of the input and output vectors at their respective time steps.
- Drop the output hypervector $\mathbf{y}(n-1))$ because the input hypervector $\mathbf{u}(n)$ can be interpreted as including the output values.

The simplified equation is:

$$\mathbf{x}(n) = \mathrm{P}(\mathbf{x}(n-1)) + \mathbf{u}(n)$$

0.2.6 Expand repeated application of the reservoir update equation

Trace out the repeated application, starting from time $n = 0$ with an empty reservoir, $\mathbf{x}(0) = \mathbf{0}$, where $\mathbf{0}$ is the additive identity hypervector.

$n = 0 : \mathbf{x}(0) = \mathbf{0}$
$n = 1 : \mathbf{x}(1) = \mathrm{P}(\mathbf{x}(0)) + \mathbf{u}(1) = \mathrm{P}(\mathbf{0}) + \mathbf{u}(1) = \mathbf{0} + \mathbf{u}(1) = \mathbf{u}(1)$

$n = 2 : \mathbf{x}(2) = \mathrm{P}(\mathbf{x}(1)) + \mathbf{u}(2) = \mathrm{P}(\mathbf{u}(1)) + \mathbf{u}(2)$ $n = 3 : \mathbf{x}(3) = \mathrm{P}(\mathbf{x}(2)) + \mathbf{u}(3) = \mathrm{P}(\mathrm{P}(\mathbf{u}(1)) + \mathbf{u}(2)) + \mathbf{u}(3)$
$= \mathrm{P}^2(\mathbf{u}(1)) + \mathrm{P}(\mathbf{u}(2)) + \mathbf{u}(3) = \mathrm{P}^2(\mathbf{u}(1)) + \mathrm{P}^1(\mathbf{u}(2)) + \mathrm{P}^0(\mathbf{u}(3))$

This is a standard VSA idiom for representation of a sequence of values (Kleyko, Rachkovskij, et al. 2022, eqn. 17). The representation is equivalent to a set of predictors, each of which is a lagged copy of the input values. For example, the $\mathrm{P}^2(\mathbf{u}())$ term represents the value of the input two time steps earlier. All the permutations are orthogonal, so the permutations effectively constitute the identities of the predictor variables. This means that the integer Echo State Network is restricted to modelling the outputs as weighted sums of the lagged inputs.

A somewhat subtle point is that this reservoir update is not directly representing the sequence $A \rightarrow B \rightarrow C$ as successor relations, rather it is indicating when the states occurred and the successor relationships are implicit in the state times.

Note that the form of the result resembles a polynomial with the powers of the pemutation corresponding to the powers of the variable of the polynomial and the input values corresponding to the coefficients of the polynomial. Note also that the result is iteratively built up, one input value at a time by transforming the result so far to reflect it's updated role and bundling in the next input value.

0.3 Polynomial feature construction

0.3.1 What does Next Generation RC do?

Next Generation Reservoir Computing replaces the recurrent reservoir calculations with the direct calculation of products of delayed input values (Gauthier et al. 2021, fig. 1). This is justified by a universal approximator result showing that the NGRC is equivalent to traditional RC. The NGRC authors see the advantage of this as being the elimination of the reservoir, in particular, the random matrix implementing the reservoir update,

From the point of view of a statistical modeller, NGRC is constructing polynomial interaction features from the input stream to use as predictors in a simple regression model. We investigate how to create equivalent VSA polynomial interaction features using VSA mechanisms.

0.3.2 RC as incremental construction of predictors

We *could* create a desired set of polynomial predictors in one step, but this would require knowing which specific predictors are required and would require dedicated VSA circuitry for each predictor to be constructed. Instead, we will use a reservoir update circuit to construct *all* the polynomial predictors incrementally.

Note that the NGRC objection to the reservoir was actually to the random matrix. The VSA reservoir update does not use a random matrix so we don't have the tuning problems identified by the NGRC authors.

VSA reservoir update relies on superposition and distributivity to construct many predictors in parallel, so it requires much less hardware than dedicated hardware for each predictor.

Also, note that because of the incremental update the predictor terms are constructed on a specific order. The simplest terms are constructed first and more complex terms later (with smaller magnitude). This constitutes an inductive bias to use lower order terms.

0.4 VSA polynomial reservoir update

0.4.1 Develop polynomial update formula

0.4.1.1 $(x+1)^n$

Base the first attempt at the reservoir update on the polynomial $(x+1)^n$. Take the simplified update equation (where $\times$ is hypervector multiplication (binding)):

$$\mathbf{x}(n) = (\mathbf{x}(n-1) + \mathbf{1}) \times (\mathbf{u}(n) + \mathbf{1})$$

Trace out the repeated application of the reservoir update equation, starting from time $n = 0$ with an empty reservoir, $\mathbf{x}(0) = \mathbf{0}$.

$\mathbf{x}(0) = \mathbf{0}$
$\mathbf{x}(1) = (\mathbf{x}(0) + \mathbf{1}) \times (\mathbf{u}(1) + \mathbf{1}) = (\mathbf{0} + \mathbf{1}) \times (\mathbf{u}(1) + \mathbf{1}) = \mathbf{u}(1) + \mathbf{1}$
$\mathbf{x}(2) = (\mathbf{x}(1) + \mathbf{1}) \times (\mathbf{u}(2) + \mathbf{1}) = (\mathbf{u}(1) + \mathbf{1} + \mathbf{1}) \times (\mathbf{u}(2) + \mathbf{1})$
$= \mathbf{u}(1) \times \mathbf{u}(2) + 2 \cdot \mathbf{u}(2) + \mathbf{u}(1) + 2 \cdot \mathbf{1}$
$\mathbf{x}(3) = (\mathbf{x}(2) + \mathbf{1}) \times (\mathbf{u}(3) + \mathbf{1})$
$= (\mathbf{u}(1) \times \mathbf{u}(2) + 2 \cdot \mathbf{u}(2) + \mathbf{u}(1) + 2 \cdot \mathbf{1} + \mathbf{1}) \times (\mathbf{u}(3) + \mathbf{1})$
$= (\mathbf{u}(1) \times \mathbf{u}(2) + 2 \cdot \mathbf{u}(2) + \mathbf{u}(1) + 3 \cdot \mathbf{1}) \times (\mathbf{u}(3) + \mathbf{1})$
$= \mathbf{u}(1) \times \mathbf{u}(2) \times \mathbf{u}(3) + 2 \cdot \mathbf{u}(2) \times \mathbf{u}(3) + \mathbf{u}(1) \times \mathbf{u}(3) + 3 \cdot \mathbf{u}(3) + \mathbf{u}(1) \times \mathbf{u}(2) + 2 \cdot \mathbf{u}(2) + \mathbf{u}(1) + 3 \cdot \mathbf{1}$

where $\cdot$ is for clarity when multiplying a scalar by a hypervector.

This reservoir update forms product and cross-product terms from lagged values of the input. It retains the old terms across iterations and adds new terms at each iteration.

Note that this update probably can't be used directly with self-inverse VSA schemes (e.g. BSC or MAP) because the multiplied terms would self-cancel. This update could be probably be used with self-inverse VSA schemes if the terms were permuted at each iteration.

To a first approximation the different scalar multipliers of the terms are irrelevant because all the terms will be multiplied by scalar coefficients in the regression readout. These readout coefficients can compensate for arbitrary scalar multipliers of the terms.

The scalar multipliers determine the relative magnitudes of the hypervectors corresponding to the algebraic terms. Given that the magnitude of the reservoir hypervector will almost certainly be normalised, this means that the terms with the smallest scalar multipliers will have the smallest smallest magnitudes. This raises the possibility that terms with very small magnitudes may become effectively invisible in the noise of the VSA system.

0.4.1.2 Multiplicative identity (1) and copy arcs

The multiplicative identity ($\mathbf{1}$) is in the update formula to copy terms across iterations. The update formula above generates a $\mathbf{1}$ term as an additive component of the reservoir hypervector at every iteration. This term is uninformative with respect to variation of the predictions. Given that some of the magnitude of the reservoir hypervector is used by the $\mathbf{1}$ term, that the magnitude of the reservoir hypervector is almost certainly normalised, and that the implementation will be noisy, the presence of the $\mathbf{1}$ term reduces the information capacity of the reservoir hypervector.

It would be interesting to see if a polynomial reservoir update can be constructed without using the multiplicative identity term ($\mathbf{1}$). The key observation here is that binding with the multiplicative identity term is effectively a copy operation in the data VSA flow graph. Translate the reservoir update equation $\mathbf{x}(n) = (\mathbf{x}(n-1) + \mathbf{1}) \times (\mathbf{u}(n) + \mathbf{1})$ to a VSA data flow graph.

The circle nodes correspond to VSA operators: binary addition and multiplication, unary delay, and nullary constant value. The arrows between nodes correspond to transfers of hypervectors. Some of the arrows are labelled with the corresponding terms in the update equation. The blue arrows correspond to the $(\mathbf{u}(n) + \mathbf{1})$ term and the red arrows correspond to the $(\mathbf{x}(n-1) + \mathbf{1})$ term, which will be simplified later.

The purpose of the $\mathbf{1}$ in $(\mathbf{u}(n) + \mathbf{1})$ is to copy $\mathbf{x}(n-1)$ into the result. This means that the updated result starts as a copy of the previous value of the result.

The purpose of the $\mathbf{1}$ in $(\mathbf{x}(n-1) + \mathbf{1})$ is to copy $\mathbf{u}(n)$ into the result. This means that the current input is added in to the updated result.

The other term added into the result is $\mathbf{x}(n-1) \times \mathbf{u}(n)$.

In the next VSA data flow graph these three terms are calculated directly and added to form the result. This graph corresponds to the reservoir update equation: $\mathbf{x}(n) = \mathbf{x}(n-1) + \mathbf{u}(n) + \mathbf{u}(n) \times \mathbf{x}(n-1)$

Trace out the repeated application of the reservoir update equation, starting from time $n = 0$ with an empty reservoir, $\mathbf{x}(0) = \mathbf{0}$.

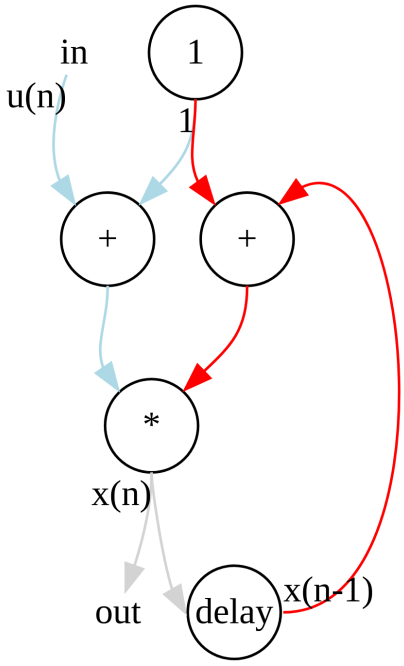

Figure 1: $\mathbf{x}(n) = (\mathbf{x}(n-1) + \mathbf{1}) \times (\mathbf{u}(n) + \mathbf{1})$

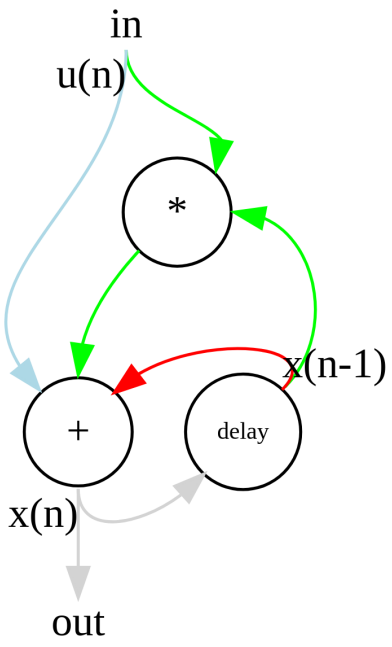

Figure 2: $\mathbf{x}(n) = \mathbf{x}(n-1) + \mathbf{u}(n) + \mathbf{u}(n) \times \mathbf{x}(n-1)$

$$\mathbf{x}(n) = \mathbf{x}(n-1) + \mathbf{u}(n) + \mathbf{u}(n) \times \mathbf{x}(n-1)$$

$\mathbf{x}(0) = \mathbf{0}$
$\mathbf{x}(1) = \mathbf{x}(0) + \mathbf{u}(1) + \mathbf{u}(1) \times \mathbf{x}(0) = \mathbf{u}(1)$
$\mathbf{x}(2) = \mathbf{x}(1) + \mathbf{u}(2) + \mathbf{u}(2) \times \mathbf{x}(1)$
$= \mathbf{u}(1) + \mathbf{u}(2) + \mathbf{u}(2) \times \mathbf{u}(1)$ $\mathbf{x}(3) = \mathbf{x}(2) + \mathbf{u}(3) + \mathbf{u}(3) \times \mathbf{x}(2)$
$= (\mathbf{u}(1) + \mathbf{u}(2) + \mathbf{u}(2) \times \mathbf{u}(1)) + \mathbf{u}(3) + \mathbf{u}(3) \times (\mathbf{u}(1) + \mathbf{u}(2) + \mathbf{u}(2) \times \mathbf{u}(1)) = \mathbf{u}(1) + \mathbf{u}(2) +$
$\mathbf{u}(2) \times \mathbf{u}(1) + \mathbf{u}(3) + \mathbf{u}(3) \times \mathbf{u}(1) + \mathbf{u}(3) \times \mathbf{u}(2) + \mathbf{u}(3) \times \mathbf{u}(2) \times \mathbf{u}(1))$

This generates all the interactions of increasing order.

0.4.1.3 Reservoir update with permutation

I noted earlier that permutation would be required if we tried to use a self-inverse VSA scheme with this kind of reservoir update that generates products of terms. Note that the interaction terms generated above are unordered because of the commutativity of the product operator. If we used a noncommutative product then we wouldn't need to use permutation to capture the the ordering of the terms.

An interesting case occurs when we use Fractional Power Encoded scalar values as inputs. FPE encoded values are equivalent to vectors and the product of FPE encoded values is equivalent to vector addition of the component vectors. The result is the same regardless of the order in which the vectors are added. If the order of the vector values is important for the problem being solved by the reservoir computer (e.g. in navigation where the path to the destination is important, rather than the location of the destination) then we need to encode the vectors in a way that is sensitive to the order, for example by applying permutation powers to the vectors.

There is probably no single "right" way to construct a reservoir update and we will need to design it to meet the requirements of the current task. Let's do that here. We'll take the previous reservoir update equation and modify it to meet some (ambiguous and ill-defined) objectives.

We will assume that the input stream is FPE-encoded vectors and that the sequence of vectors is a reasonable predictor of the outcome. We will also assume that polynomial terms of the predictors are important predictors.

The previous update equation created output terms that were direct copies of prior input terms. This is effectively claiming that input values are good predictors regardless of when they occurred in the past. Let's assume that is likely to be incorrect, so we need to modify the update equation to prevent the repeated direct copying of inputs across generations. We attempt to do this by deleting the arrow between the delay node and the addition node.

We believe the successor relation is predictive, so need to bind successor states and apply the "before" permutation P_{bef} to the representation of the earlier state. We do this by inserting the permutation node P_{bef} into the arrow between the delay node and the product node.

This graph corresponds to the reservoir update equation: $\mathbf{x}(n) = \mathbf{u}(n) + \mathbf{u}(n) \times \mathrm{P}_{bef}(\mathbf{x}(n-1))$ That is, the current state consists of the current input plus the encoded sequence of states leading to the current state.

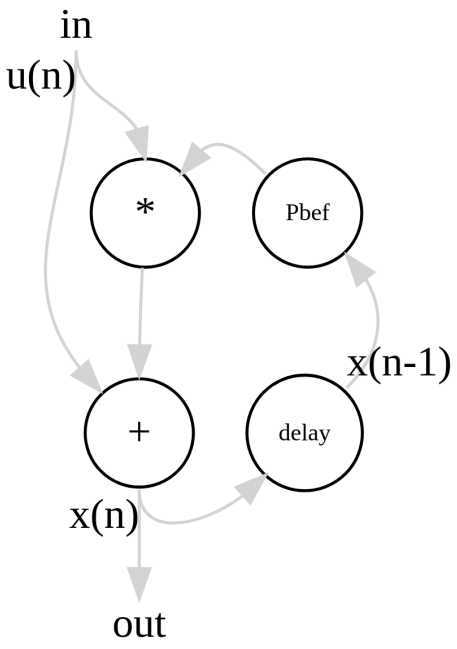

Figure 3: $\mathbf{x}(n) = \mathbf{u}(n) + \mathbf{u}(n) \times \mathrm{P}_{bef}(\mathbf{x}(n-1))$

Trace out the repeated application of the reservoir update equation, starting from time $n = 0$ with an empty reservoir, $\mathbf{x}(0) = \mathbf{0}$.

$\mathbf{x}(n) = \mathbf{u}(n) + \mathbf{u}(n) \times \mathrm{P}_{bef}(\mathbf{x}(n-1))$

$\mathbf{x}(0) = \mathbf{0}$
$\mathbf{x}(1) = \mathbf{u}(1) + \mathbf{u}(1) \times \mathrm{P}_{bef}(\mathbf{x}(0)) = \mathbf{u}(1)$
$\mathbf{x}(2) = \mathbf{u}(2) + \mathbf{u}(2) \times \mathrm{P}_{bef}(\mathbf{x}(1))$
$= \mathbf{u}(2) + \mathbf{u}(2) \times \mathrm{P}_{bef}(\mathbf{u}(1))$
$\mathbf{x}(3) = \mathbf{u}(3) + \mathbf{u}(3) \times \mathrm{P}_{bef}(\mathbf{x}(2))$
$= \mathbf{u}(3) + \mathbf{u}(3) \times \mathrm{P}_{bef}(\mathbf{u}(2) + \mathbf{u}(2) \times \mathrm{P}_{bef}(\mathbf{u}(1)))$
$= \mathbf{u}(3) + \mathbf{u}(3) \times \mathrm{P}_{bef}(\mathbf{u}(2)) + \mathbf{u}(3) \times \mathrm{P}_{bef}(\mathbf{u}(2) \times \mathrm{P}_{bef}(\mathbf{u}(1)))$

$$= \mathbf{u}(3) + \mathbf{u}(3) \times \mathrm{P}_{bef}(\mathbf{u}(2)) + \mathbf{u}(3) \times \mathrm{P}_{bef}(\mathbf{u}(2)) \times \mathrm{P}^2_{bef}(\mathbf{u}(1)))$$

This reservoir generates all the sequences of input values of increasing length.

Obviously there is plenty of scope for modifying the update equation to generate different predictors.

0.5 Feature fading

In general every iteration through the reservoir update equation can add more terms to the reservoir state. This can continue arbitrarily long, so the reservoir state can become arbitrarily complex (when viewed as algebraic terms).

You might think this would cause problems because the number of features keeps increasing and because it implies infinite memory (terms hanging around forever). In practice this is not an issue because the magnitudes of the terms decrease over time because of the bundling update. Eventually the terms will have such a low magnitude that they are effectively lost in the system noise.

0.5.1 Nonassociative bundling

Bundling (as implemented, rather than as a mathematical abstraction) loses information. The more times a term has passed through bundling, the more information it has lost. This is reflected in the reducing magnitude of the hypervector representing the algebraic term.

This is most obvious when you consider randsel bundling (Gayler 2022; Heddes et al. 2022). In randsel bundling each output element is selected at random from the corresponding elements of the argument hypervectors. With two arguments and equal selection probability only half the elements are copied from each argument hypervector to the result hypervector. Consequently, the hypervector representing each algebraic term has half it's original magnitude. Thus, the half-life of each term in the reservoir would be one time step.

0.5.2 Weighted bundling

Note that the bundling operator doesn't have to be binary, it can have an arbitrary (but generally small) number of arguments. We can also associate a positive scalar weight with each argument that controls the relative contribution of that argument to the result.

For example in Figure 2, the bundling operator has three arguments. This could be implemented as weighted bundling and the weights used to control the relative magnitudes of the arguments in the result. Increasing the weight of the $\mathbf{x}(n-1)$ term would increase the magnitude of the current reservoir state preserved in the next state and therefore extend the

retention duration of the reservoir (at the cost of decreasing the relative magnitudes of the $\mathbf{u}(n)$ and $(\mathbf{u}(n) \times \mathbf{x}(n-1))$ added in at each time step).

0.5.3 Bundled inputs

The preceding discussions might have given the impression that the input stream ($\mathbf{u}(n)$) consists of atomic vectors. However, the joy of VSA is that everything is just a hypervector value, so the input stream could be arbitrarily complicated values. In particular the input stream could consist of the sum of the input and output streams used in classical RC, and wighted bundling could be used to control their relative contributions.

0.6 Development suggestions

The following points are directions that might be worth investigating.

0.6.1 Input/hierarchical encoding

It might be advantageous to construct sequence representations in the input streams rather than in the reservoir update. This might be equivalent to hierarchical reservoirs. We could have input-stream-specific reservoirs that create input sequence representations and a stream fusion reservoir (or maybe stream fusion doesn't need the recurrent update).

0.6.2 Intercept term

Reservoir readout is performed by a weighted sum (effectively a regression model). Statistical regression normally allows for an intercept (bias) term and the goodness of fit can depend strongly on whether an intercept term is included in the readout model.

Typical statistical regression software normally constructs the intercept term implicitly rather than requiring it to be explicitly present as a predictor. Depending on how the readout has been implemented in some VSA RC model it is possible that the intercept term has not been included.

- For any specific VSA RC implementation, check on how the intercept term is treated in the reservoir readout (both in training and prediction).
- If readout is performed by standard regression (including ridge regression) it can be implemented by concatenating a single constant 1 element to the RC reservoir hypervector.
- If readout is implemented by VSA bundling it is only necessary to ensure that the reservoir hypervector contains a multiplicative identity hypervector term ($\mathbf{1}$).

0.6.3 Bundling and capacity

The discussion of feature fading and weighted bundling shows that we need a better treatment of bundling and capacity than just considering the number of items that are included in the bundle. It's not just the number of superposed items, which can be changed arbitrarily by algebraic re-arrangement. Also, a D-dimensional HRR binding is equivalent to the sum of D-many MAP bindings, but only counts as one item in the traditional capacity analysis.

0.6.4 Warm up

Standard RC uses a warm-up to allow the reservoir state to get past any transient states arising from the initial value of the reservoir.

- Investigate the impact of initial values on the length of required warm-up. In the formulae above I used zero hypervector initial values. Other possibilities are the one hypervector and a randomly selected hypervector. Another reasonable choice might be to randomly select a value from the set of reservoir values seen after warm-up. I would be interesting to investigate how long it takes the reservoir to synchronise to the new input.
- Investigate whether it is possible to read-out from the reservoir during the warm-up phase. A realistic environment is non-stationary, so arguably there is a continual stream of (partial?) re-starts. There would be an evolutionary advantage to the readout being useful, regardless of how recently the system had been restarted.

0.6.5 Dynamic readout

Rather than having a fixed-weight readout mechanism, is it possible to have a readout mechanism that varies in response to the task demands at readout time? This might be related to avoidance of catastrophic forgetting (e.g. French (1994)) and generalisation at recall rather than in a separate training phase (e.g. Shabahang, Yim, and Dennis (2022)).

0.6.6 Dynamic update

Continuing the same theme of making RC mechanisms dynamic, it might be possible for the update formula to change dynamically to indefinitely preserve the contents of the reservoir as a working memory and to emphasize representational components that are most relevant to the current task. For example, it might be appropriate to insert a dynamically updated cleanup memory in the reservoir update loop.

Another point to consider is whether to include gating in the reservoir update loop, so that the reservoir is only updated when there are material changes in the environment rather than every time step.

Frady, E. Paxon, Denis Kleyko, and Friedrich T. Sommer. 2018. "A Theory of Sequence Indexing and Working Memory in Recurrent Neural Networks." *Neural Computation* 30 (6): 1449–1513. https://doi.org/10.1162/neco_a_01084.

French, Robert M. 1994. "Dynamically Constraining Connectionist Networks to Produce Distributed, Orthogonal Representations to Reduce Catastrophic Interference." In *Proceedings of the Sixteenth Annual Conference of the Cognitive Science Society*, edited by Ashwin Ram and Kurt Eiselt, 335–40. Atlanta, GA, USA: Routledge. https://doi.org/10.4324/9781315789354-58.

Gauthier, Daniel J., Erik Bollt, Aaron Griffith, and Wendson A. S. Barbosa. 2021. "Next Generation Reservoir Computing." *Nature Communications* 12 (1): 5564. https://doi.org/10.1038/s41467-021-25801-2.

Gayler, Ross W. 2022. "Implement Randsel Bundling · Issue #75 · Hyperdimensional-Computing/Torchhd." GitHub repository. GitHub. June 4, 2022. https://github.com/hyperdimensional-computing/torchhd/issues/75.

Heddes, Mike, Igor Nunes, Dheyay Desai, and Pere Vergés. 2022. "Randsel — Torchhd Documentation." Torchhd Documentation. 2022. https://torchhd.readthedocs.io/en/stable/generated/torchhd.randsel.html.

Kanerva, Pentti. 1997. "Fully Distributed Representation." In, 358–65.

Kleyko, Denis, Edward Paxon Frady, Mansour Kheffache, and Evgeny Osipov. 2022. "Integer Echo State Networks: Efficient Reservoir Computing for Digital Hardware." *IEEE Transactions on Neural Networks and Learning Systems* 33 (4): 1688–1701. https://doi.org/10.1109/TNNLS.2020.3043309.

Kleyko, Denis, Dmitri A. Rachkovskij, Evgeny Osipov, and Abbas Rahimi. 2022. "A Survey on Hyperdimensional Computing Aka Vector Symbolic Architectures, Part I: Models and Data Transformations." *ACM Computing Surveys*, May, 3538531. https://doi.org/10.1145/3538531.

Qiu, Frank. n.d. "Graph Embeddings via Tensor Products and Approximately Orthonormal Codes." http://arxiv.org/abs/2208.10917.

Shabahang, Kevin D., Hyungwook Yim, and Simon J. Dennis. 2022. "Generalization at Retrieval Using Associative Networks with Transient Weight Changes." *Computational Brain & Behavior* 5 (1): 124–55. https://doi.org/10.1007/s42113-022-00127-4.