Learning to Predict in Networks with Heterogeneous and Dynamic Synapses

Daniel Burnham

A Master's thesis
submitted in partial fulfillment of the
requirements for the degree of

Master of Science

University of Washington

2021

Committee:

Eric Shea-Brown

Stefan Mihalas

Adrienne Fairhall

Program Authorized to Offer Degree:

Applied Mathematics

University of Washington

Abstract

Learning to Predict in Networks with Heterogeneous and Dynamic Synapses

Daniel Burnham

Chair of the Supervisory Committee:
Prof. Eric Shea-Brown
Applied Mathematics

A salient difference between artificial and biological neural networks is the complexity and diversity of individual units in the latter [36]. This remarkable diversity is present in the cellular and synaptic dynamics. In this study we focus on the role in learning of one such dynamical mechanism missing from most artificial neural network models, short term synaptic plasticity (STSP). Biological synapses have dynamics over at least two time scales: a long time scale, which maps well to synaptic changes in artificial neural networks during learning, and the short time scale of STSP, which is typically ignored. Recent studies have shown the utility of such short term dynamics in a variety of tasks [28, 29], and networks trained with such synapses have been shown to better match recorded neuronal activity and animal behavior [18]. Here, we allow the timescale of STSP in individual neurons to be learned, simultaneously with standard learning of overall synaptic weights. We study learning performance on two predictive tasks, a simple dynamical system and a more complex MNIST pixel sequence. When the number of computational units is similar to the task dimensionality, RNNs with STSP outperform standard RNN and LSTM models. A potential explanation for this improvement is the encoding of activity history in the short term synaptic dynamics, a biological form of long short term memory. Beyond a role for synaptic dynamics themselves, we find a reason and a role for their diversity: learned synaptic time constants become heterogeneous across training and contribute to improved prediction performance

in feedforward architectures. These results demonstrate how biologically motivated neural dynamics improve performance on the fundamental task of predicting future inputs with limited computational resources, and how learning such predictions drives neural dynamics towards the diversity found in biological brains.

TABLE OF CONTENTS

LIST OF FIGURES

GLOSSARY

ARTIFICIAL NEURAL NETWORK: a collection of connected units or nodes called artificial neurons, which loosely model the neurons in a biological brain.

DYNAMICS: sequences describing how a point in a geometrical space change in time. An example of a physical system with temporal dynamics is the swinging of a clock pendulum.

FILTER: a device or process that removes some unwanted components or features from a signal. Can also perform computation on an input signal.

IMPULSE RESPONSE: a system output when presented with a brief input signal, called an impulse.

NONLINEAR: a system in which the change of the output is not proportional to the change of the input.

OPTIMIZATION: selection of a best element, with regard to some criterion, from some set of available alternatives.

PREDICTION: formulating an expectation for the future behavior of a system.

REGRESSION: a set of statistical processes for estimating the relationships between a dependent variable and one or more independent variables.

REGULARIZATION: the process of adding information in order to solve an ill-posed problem or to prevent overfitting.

STOCHASTIC: randomly determined; having a random probability distribution or pattern that may be analyzed statistically but may not be predicted precisely.

SYNAPSE: a junction between two nerve cells, consisting of a minute gap across which impulses pass by diffusion of a neurotransmitter.

SYNAPTIC PLASTICITY: changes in synaptic strength between a pre and postsynaptic neuron in response to stimuli due to biophysical dynamics that occur on multiple time scales.

TIME SERIES: a series of data points indexed in time order. Temperature measurements taken throughout the day would form a time series.

RECURRENCE: in the context of artificial neural networks, connections that occur between neurons horizontally within a layer and between a given neuron and itself. Recurrent connections serve to construct closed loops of connectivity within the artificial neural network layer.

ACKNOWLEDGMENTS

First, I would like to express a deep gratitude to my parents for encouraging me to follow my interests and supporting my meanderings across numerous career paths. They instilled in me a strong sense of curiosity and wonder that fuels me to this day.

Secondly, my spouse, Daniela, has helped me to establish a balance between work and life and to maintain a soft focus on the broader picture. She brings joy to each day and makes me excited about what the future holds. Her interest in my research has been extremely helpful as on many occasions our conversations have highlighted topics that I did not quite understand, or problems with my approach that I had not considered. I am beyond fortunate to have such a brilliant and encouraging partner.

Lastly, I am extremely grateful to the computational and theoretical neuroscience community at the University of Washington and the Allen Institute. Reading the work coming from the UW Computational Neuroscience Center and the Allen Institute for Brain Science inspired my career change in 2019 and specifically drew me to the Applied Mathematics Department at UW. I still cannot believe that I have the opportunity to work with such incredible scientists and human beings. More specifically, I am thankful for the support, kindness, and mentorship of Roman Levin, Matt Farrell, Gabrielle Gutierrez, Doris Voina, Stefano Recanatesi, and Leenoy Meshulam. In addition, I am indebted to the members of my thesis committee for their guidance, insight, and exemplary leadership. To Eric: thank you for your wonderfully exciting courses, for your encouragement and for the focus and energy you bring to every conversation. To Stefan: thank you for your excitement, your creative solutions, and your eagerness to teach. To Adrienne: thank you for creating a sense of belonging for me in your research group, for your insights during my lab meeting presentations, and for supporting me as a member of my master's thesis committee. I feel extremely privileged to have this opportunity to pursue science that I find meaningful and to have such a wonderful group of collaborators, mentors, and friends.

DEDICATION

to my parents

Chapter 1

INTRODUCTION

1.1 Motivation

Predicting dynamics is a fundamental requirement of intelligent behavior. For living systems, prediction of temporal dynamics is particularly important given the myriad of environmental dynamics present within Earth's ecosystems. Observing and responding to these environmental dynamics in many cases is integral to the adaptation of a particular organism to its niche. For this reason neural systems have been theorized to be driven by prediction of the external dynamics across a hierarchy of sensory signal processing [30]. This makes dynamics prediction an important paradigm in which to study neural system function. Moreover, comparisons between biological and artificial neural networks have begun to elucidate characteristics of biological networks that may be important differentiators of neurobiological dynamics prediction performance. One of the most salient observed differences between biological and artificial neural networks is the heterogeneity of neuronal features showing remarkably diverse anatomy [14], gene expression [35], intrinsic dynamics [37], *in vivo* responses [7], and, our focus here, remarkably diverse synaptic properties [33]. This thesis focuses on the role in learning of one such dynamical mechanism missing from most artificial neural network models, short term synaptic plasticity (STSP). Biological synapses have dynamics over at least two time scales: a long time scale, which maps well to synaptic changes in artificial neural networks during learning, and the short time scale of STSP, which is typically ignored. Recent studies have shown the utility of such short term dynamics in a variety of tasks [28, 29], and networks trained with such synapses have been shown to better match recorded neuronal activity and animal behavior [18]. The aim of this research is to demonstrate how biologically motivated neural dynamics improve performance on the fun-

damental task of predicting future inputs, and how learning such predictions drives neural dynamics towards the diversity found in biological brains.

1.2 Background

1.2.1 Time Series Prediction

In order to understand the role of intrinsic dynamics in prediction, it is important to understand how time series prediction is accomplished in theory. To predict is to produce a model of the past to generate expectations of the future. In principle, any model can be used by a system to generate expectations, but the prediction accuracy will depend on how well the model maps information at past time steps to the future. Before one can consider how a model accomplishes this mapping, the signal to predict must have certain properties consistent with being a "predictable" signal. For the next time step of a stochastic process to be predictable for a given time step x_n, the previous time step x_{n-1} must be knowable (i.e. the process must be measurable) [5]. However this does not alone guarantee that a quality prediction can be made for the time series. Indeed, most real word signals exist on a spectrum between stochastic and deterministic. In the case of a purely stochastic process, individual random events are unpredictable, but if the probability distribution is known, the frequency of different outcomes over repeated events is predictable. Brownian motion is an example of a stochastic process where the state at the next time step is independent of what happened at all previous time steps. In this way, Brownian motion is an example of an unpredictable signal. On the other end of the spectrum, a completely deterministic signal is one dictated by a physical law or mathematical function where there is no uncertainty about the evolution of the signal in time.

Linear Filters

There are traditional methods of time series analysis that can be used to find suitable models to explain the data and in turn can be used to predict the future of a time series. These

are autoregressive and moving average models described in the domain of signal processing. Signal processing is in general most concerned with linear, translation-invariant operations on data series. These operations are implemented by filters. Filters operate on an input sequence r, producing an output sequence $\hat{y}$. There are two basic filter architectures, known as the finite impulse response (FIR) filter (Equation 1.1) and the infinite impulse response (IIR) filter (Equation 1.2). These filter types are distinct due to how each responds to an impulse input. FIR filter impulse responses decay to zero in finite time where as IIR filter impulse responses may respond indefinitely due to the incorporation of local recurrence (Figure 1.1).

$$\hat{y}(n) = \sum_{i=0}^{N} \beta_i r(n-i) + \epsilon(n) \tag{1.1}$$

$$\hat{y}(n) = \sum_{i=0}^{N} \beta_i r(n-i) - \sum_{i=0}^{M} \alpha_i \hat{y}(n-i) + \epsilon(n) \tag{1.2}$$

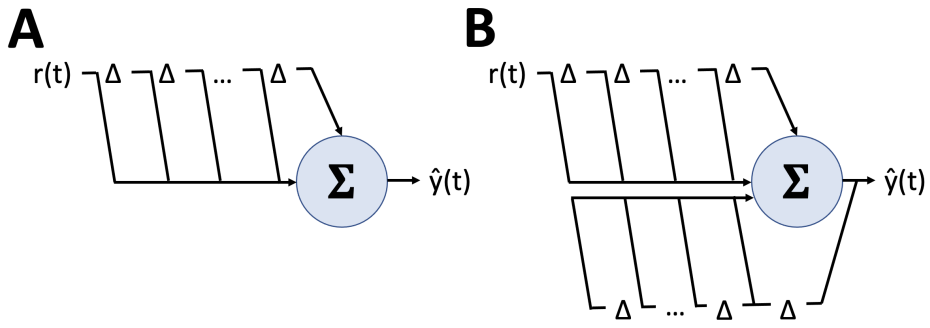

Figure 1.1: **A** FIR filter schematic with the number of time steps included in the summation dictating the order of the filter [2]. **B** IIR filter schematic which includes feedback from the output values [2].

Time series analysis would describe the operation of a FIR filter as a weighted moving average with a window size dictated by how many past time points are included in the average calculation. Through the feedback, an IIR filter adds dependency on previous output time

steps, which is defined as autoregression. Overall, an IIR filter implements an autoregressive moving average (ARMA) model of the input with the autoregressive component dependent on the feedback coefficients and the moving average component directed by the feedforward coefficients.

Coefficient Optimization

Having established the predictability of the signal and the fundamental time series analysis operations, the next step is to adapt the filter operations to make predictions of future time steps. This adaptation can occur through mathematical optimization of the filter coefficients (e.g. the β_i's and α_i's of Equations 1.1 and 1.2). One standard approach to fitting these coefficients for linear adaptive filters is linear regression. Regression models arise as solutions to the system of equations $AX = B$ where the values of A represent the input data and B represents the output values which in the context of time series prediction would correspond to future time series values. The matrix X therefore represents the mapping of the data in A to the outputs in B. In the filter coefficient optimization context, the number of coefficients corresponds to the number of previous time steps and the number of previous outputs (in the case of the IIR filter) that are used to make a prediction for future time series values. In this setting, the number of rows of the data matrix A is the number of sequences within the overall time series to use as examples to optimize the filter coefficients. The column dimension of A is then the number of coefficients. Depending on how many time series sequences are available and/or used for optimization, it may be the case that the regression problem is either over or underdetermined. This means that the data in A has either more instances (rows) than parameters (columns) or vice versa. In the case of an overdetermined system there are more equations than unknowns resulting in a system of equations that do not agree on a solution X. In the underdetermined case there are fewer equations than unknowns meaning that there are either zero solutions or infinitely many. In either case, a constraint or penalty is needed to settle on a single solution. This constraint is termed regularization and defines the regression optimization problem. The underlying

optimization objective functions for finding a linear map in either an overdetermined (1.3) or underdetermined (1.4) case are given by [4]:

$$\arg\min_X \ (\|AX - B\|_2 + \alpha g(X)) \tag{1.3}$$

$$\arg\min_X g(X) \text{ subject to } \|AX - B\|_2 \leq \epsilon \tag{1.4}$$

where $g(X)$ is a given penalization, with penalty parameter α, in the case of an overdetermined system (Equation 1.3). The term $g(X)$ is of particular interest because it can be used to modulate the characteristic(s) of the solution X that is sought in solving the linear systems of equations $AX = B$ through optimization. The regularization term $g(X)$ can take on different forms depending on the type of regression:

$$\textbf{Ordinary Least Squares: } \arg\min_X (\|AX - B\|_2^2) \tag{1.5}$$

$$\textbf{Ridge: } \arg\min_X (\|AX - B\|_2^2 + \alpha \|X\|_2^2) \tag{1.6}$$

$$\textbf{Lasso: } \arg\min_X (\frac{1}{2n} \|AX - B\|_2^2 + \alpha \|X\|_1) \tag{1.7}$$

It is evident in each case (Equations 1.5-1.7) that the regularization is distinct. Ordinary least squares does not include a penalty at all. Ridge regression adds a penalty based on the ℓ_2 of X. Lasso regression penalizes the solution based on the ℓ_1 norm. Solving the least squares optimization problem given a set of filter input and output data would allow for a set of filter coefficients to be determined such that the filter operation best produces a future state prediction.

Nonlinear Filters

As indicated previously, this procedure of determining filter operation coefficients for time series prediction works for LTI systems. In practice, there are many instances where the linearity condition is violated. A system is linear if, for a given scalar multiplied input $ar(t)$,

the corresponding output is likewise scaled by the same factor a. Linear systems also exhibit the superposition property where the output yielded for two or more inputs is equivalent to the sum of each output resulting from the inputs separately. A nonlinear signal does not have these properties and in turn cannot be adequately predicted using linear filter operations. Perhaps intuitively, to address these instances where prediction is desired for signals with nonlinear behavior, nonlinear filter operations can be used. Polynomial filters are a nonlinear generalisation of linear filters [40, 22]. Polynomial filters are often referred to as Volterra filters when they are based on the Volterra series [40, 22]. The Volterra series is an extension of the Taylor series expansion for representing nonlinear systems. An additional feature is that the Volterra series allows for nonlinear dependency on the input for past time steps [40, 22].

$$\begin{aligned}\hat{y}(n) = h_0 + \sum_{\tau_1=a}^{b} h_1(\tau_1) r(n-\tau_1) + \sum_{\tau_1=a}^{b}\sum_{\tau_1=a}^{b} h_1(\tau_1,\tau_2) r(n-\tau_1) r(n-\tau_2) \\ + \sum_{\tau_1=a}^{b}\sum_{\tau_2=a}^{b} ... \sum_{\tau_p=a}^{b} h_p(\tau_1,\tau_2,...,\tau_p) r(n-\tau_1) r(n-\tau_2) ... r(n-\tau_p) + ...\end{aligned} \tag{1.8}$$

$$\hat{y}(n) = h_0 + \sum_{p=0}^{P}\sum_{\tau_1=a}^{b} ... \sum_{\tau_p=a}^{b} h_p(\tau_1,\tau_2,...,\tau_p) \prod_{p}^{j=1} r(n-\tau_j) \tag{1.9}$$

In Equation 1.9, $h_p(\tau_1, \tau_2, ..., \tau_p)$ are termed the discrete time Volterra kernels and act as weighting coefficients. If P is finite, as is the case in practice, the series operator is said to be truncated. If a, b and P are finite, the series operator is called a doubly finite Volterra series. If $a \geq 0$, the operator is said to be causal.

1.2.2 Artificial Neural Networks

Equipped with a picture of how time series prediction can be implemented theoretically, an interesting next step is to describe the connection between filter operations for time series analysis and artificial neural network (ANN) operations. To begin, an artificial neural network is a model that can range from loosely inspired by neurobiology to highly comparable

to the biophysics of biological neural circuitry. In the most general sense, an ANN contains a collection of nodes, analogous to neurons, and connections between these nodes which are analogous to synapses. The collection of these nodes and connections can be organized into various architectures. Key architectural features include the number and organization of layers of nodes and the direction of the connections between nodes. For a given pair of nodes, connections between them can be either feedforward, meaning unidirectional where one node simply feeds directly into the other, or recurrent, meaning bidirectional where both nodes are connected to one another reciprocally as well as wired to feedback on themselves. These two connection styles can be implemented similarly in larger networks of more nodes and layers. The last decades have witnessed astonishing development in neural network research, both in artificial intelligence (AI) applications to problems across the sciences and in creating model systems for understanding computation in the brain. The essential concepts underlying these advances have their origin in neuroscience: distributed computing using neurons, learning by changing synaptic connections, and hierarchical organization of networks [9, 20]. Likewise, incorporation of these biological features into AI models, together with analyses of emerging large-scale neural datasets, has allowed these models to inform the underlying biology.

Neural Networks as Filters

If we observe the case of a single, feedforward linear neural network node, we can see that the operation which it carries out on the input is similar to that of the FIR filter in Section 1.2.1. For a given discrete time input $r(n)$ the weights, w_i can be optimized to predict the next time step $r(n+1)$:

$$\hat{y}(t) = \hat{r}(n+1) = r(n)w_i + b \tag{1.10}$$

$$\hat{y}(n) = \hat{r}(n+1) = \sum_{i=0}^{N=0} \beta_i r(n-i) + \epsilon(n) \tag{1.11}$$

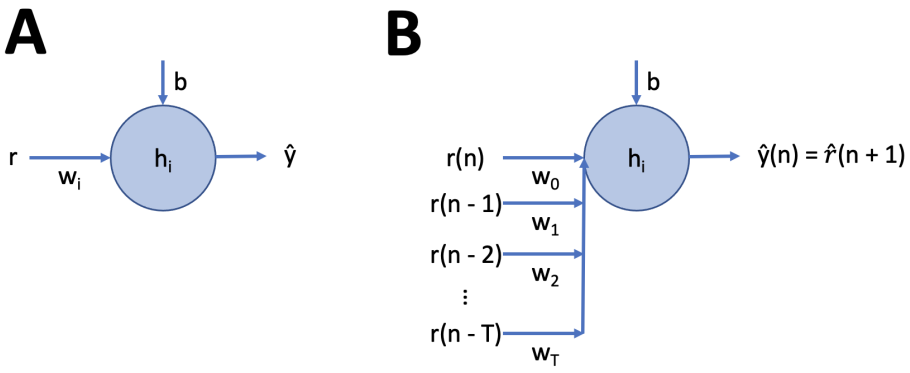

Figure 1.2: **A** Single linear node with input r, input weight w_i, bias b, and output value or predicted value $\hat{y}$. **B** Single linear node with time series embedding.

Here $\hat{y}(t)$ and $\hat{r}(n+1)$ both correspond to the output of the filter as a predicted value in comparison to $y(t)$ and $r(n+1)$ which are the actual time series value. This operation corresponds to an order 1 FIR filter, in that only a single past point in time is convolved with a single coefficient to produce a response. For complex linear signals, an order 1 FIR filter may be a poor model (Equation 2.6). To improve the model, the time series r can be embedded such that successive values of the time series are retained and treated as additional spatial dimensions in the input space (Figure 1.2B). This corresponds to setting the value of N in Equation 1.1 to any value greater than 1, to combine $N+1$ coefficients with past time points of input sequence r back to $r(n-T)$.

Next, observe the case of a single, recurrent linear neural network node (Figure 1.3A). The equation describing the output for this node (Equation 1.12) demonstrates that the local recurrence yields a node that operates as an IIR filter (Equation 1.13).

$$\hat{y}(n) = \hat{r}(n+1) = r(n)w_i + \hat{y}(n)q_i + b \tag{1.12}$$

$$\hat{y}(n) = \hat{r}(n+1) = \sum_{i=0}^{N=0} \beta_i r(n-i) + \sum_{i=0}^{K=0} \alpha_i \hat{y}(n-i) + \epsilon(n) \tag{1.13}$$

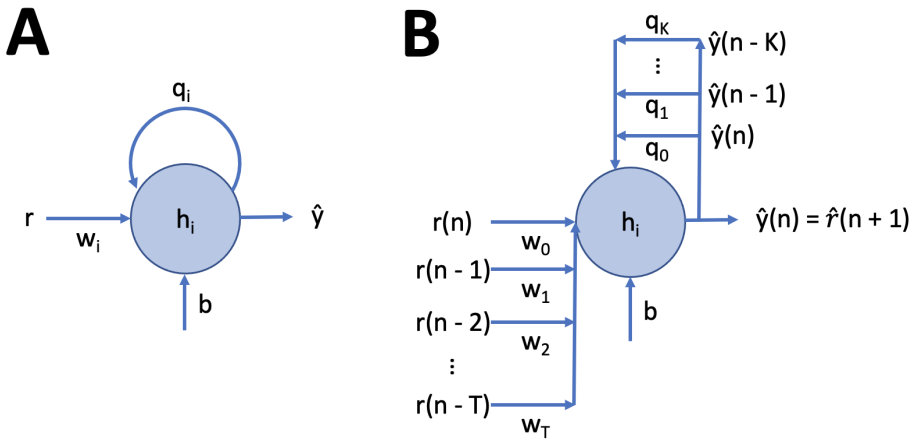

Figure 1.3: **A** Single recurrent linear node with input r, input weight w_i, bias b, recurrent weight q_i and output value or predicted value $\hat{y}$. **B** Single recurrent linear node with time series embedding for both the input and output.

These cases are simplified not only due to illustrating a single node, but also because the outputs depend linearly on the inputs. This is not generally the case in ANN models and is certainly not true of biological neurons. Indeed it is common for the output of a ANN node to pass through an activation function nonlinearity. Usually the nonlinearities that are used as activation functions are chosen such that they are bounded, monotonically increasing, and differentiable. The hyperbolic tangent function is a canonical example of such a function with a Taylor series expansion shown below:

$$y = tanh(x) = \frac{e^{2x} - 1}{e^{2x} + 1} \tag{1.14}$$

$$y = \frac{2x + \frac{(2x)^2}{2!} + \frac{(2x)^3}{3!} + \frac{(2x)^4}{4!} + O(x^5)}{2 + 2x + \frac{(2x)^2}{2!} + \frac{(2x)^3}{3!} + \frac{(2x)^4}{4!} + O(x^5)} \tag{1.15}$$

Another common activation function is the rectified linear activation function (ReLU) which is a piecewise linear function that outputs the input directly if it is positive, otherwise, it will output zero.

If we assume a single neuron model like Figure 1.2 with a hyperbolic tangent activation

function, the neuron would no longer behave as an FIR filter because the output will be composed of an infinite order combination of the summed and weighted inputs [22]. This can be made clear by writing the full equation for the node output as:

$$\hat{y}(n) = \frac{2\left(r(n)w_i + b\right) + \frac{(2(r(n)w_i+b))^2}{2!} + \frac{(2(r(n)w_i+b))^3}{3!} + \frac{(2(r(n)w_i+b))^4}{4!}}{2 + 2\left(r(n)w_i + b\right) + \frac{(2(r(n)w_i+b))^2}{2!} + \frac{(2(r(n)w_i+b))^3}{3!} + \frac{(2(r(n)w_i+b))^4}{4!}} \tag{1.16}$$

$$\hat{y}(n) = \frac{2\left(\sum_{i=0}^{N} r(n-i)w_i + b\right) + \frac{(2\left(\sum_{i=0}^{N} r(n-i)w_i+b\right))^2}{2!} + \frac{(2\left(\sum_{i=0}^{N} r(n-i)w_i+b\right))^3}{3!} + ...}{2 + 2\left(\sum_{i=0}^{N} r(n-i)w_i + b\right) + \frac{(2\left(\sum_{i=0}^{N} r(n-i)w_i+b\right))^2}{2!} + \frac{(2\left(\sum_{i=0}^{N} r(n-i)w_i+b\right))^3}{3!} + ...} \tag{1.17}$$

It can be observed from Equation 1.17 that as the number of weighted inputs to the node is increased, the node will more accurately approximate a Volterra kernel [22]. Using Figure 1.2B as an example, more weighted inputs translates to more past time points incorporated in the filter operation. Increasing the number of inputs, increases the number of w_i's therefore increasing the the number of higher order terms in Equation 1.17. Expanding this to a single layer, multi-node feedforward network yields:

$$\textbf{Feedforward: } h(n) = \left[\mathbf{W}^{hx} r(n) + \mathbf{b}^h\right] \tag{1.18}$$

$$\hat{y}(n) = \frac{2\left[\mathbf{W}^{hx} r(n) + \mathbf{b}^h\right] + \frac{(2\left[\mathbf{W}^{hx} r(n)+\mathbf{b}^h\right])^2}{2!} + ...}{2 + 2\left[\mathbf{W}^{hx} r(n) + \mathbf{b}^h\right] + \frac{(2\left[\mathbf{W}^{hx} r(n)+\mathbf{b}^h\right])^2}{2!} + ...} \tag{1.19}$$

Similarly, a neural network with global recurrent connections can be described as in Equation 2.11 with a second weight matrix $\mathbf{W}^{hh}$ denoting the connection strengths between hidden layer nodes.

$$\textbf{RNN: } h(n) = \left[\mathbf{W}^{hh} h(n-1) + \mathbf{W}^{hx} r(n) + \mathbf{b}^h\right] \tag{1.20}$$

$$\hat{y}(n) = \frac{2\left[\mathbf{W}^{hh} h(n-1) + \mathbf{W}^{hx} r(n) + \mathbf{b}^h\right] + \frac{(2\left[\mathbf{W}^{hh} h(n-1)+\mathbf{W}^{hx} r(n)+\mathbf{b}^h\right])^2}{2!} + ...}{2 + 2\left[\mathbf{W}^{hh} h(n-1) + \mathbf{W}^{hx} r(n) + \mathbf{b}^h\right] + \frac{(2\left[\mathbf{W}^{hh} h(n-1)+\mathbf{W}^{hx} r(n)+\mathbf{b}^h\right])^2}{2!} + ...} \tag{1.21}$$

One can observe that adding global recurrence increases the number of higher order terms in the expressions thus expanding the repertoire of time-invariant functional relationships that the network can approximate.

Recurrent Neural Networks

The RNN architecture introduced in Equation 2.11 is a powerful ANN model for learning sequential data due to the incorporation of the past hidden state term $h(t-1)$ which can represent salient input signal characteristics from the previous time step. However, when there are important signal properties that exist on longer time scales RNNs can fail to adequately learn to perform computation on the sequential data [17, 3]. This is due to how weights are updated during learning with backpropagation through time (BPTT).

Challenges of Learning with Backpropagation Briefly, backward propagation of errors is a method for optimizing weights in ANNs stemming from how well the output of the network compares to a desired output. Backpropagation operates by determining how much a network parameter, such as a connection weight, contributes to the error between the network output and the desired output. To determine this that gradient of the neural network parameters is needed, and backpropagation calculates this by stepping backwards through the network using the chain rule to determine the derivative of the loss with respect to each network parameter. Therefore, as more layers are added to a feedfoward architecture, the chain rules for determining the gradient of the network parameters in the earliest layers will have more and more multiplied values. One can imagine that if any of these components of the chain rule are sufficiently large or small, the product and thus the gradient will "explode" or "vanish". If this is the case, the updates to the corresponding network parameters will likewise be inordinately large or small. This problem in deep feedforward networks can occur in RNNs of only a single layer due to the recurrent connections dictating that the gradient for the network parameters must depend on the history of activity for all input time steps up to the current time t. To make this more concrete, take a simple model of a single layer

RNN where h_t represents the hidden state of the layer, x_t is the input to the layer, o_t is the output for time step t, and f and g are activation functions[41].

$$h_t = f(x_t, h_{t-1}, w_h) \tag{1.22}$$

$$o_t = g(h_t, w_o) \tag{1.23}$$

The network error, or the difference between the actual and desired output can be written as a loss function across all time steps of the input sequence to the network.

$$L(x_1, \ldots, x_T, y_1, \ldots, y_T, w_h, w_o) = \frac{1}{T}\sum_{t=1}^{T} l(y_t, o_t) \tag{1.24}$$

Now computing the gradient of the loss with respect to the hidden weight parameters yields:

$$\frac{\partial L}{\partial w_h} = \frac{1}{T}\sum_{t=1}^{T} \frac{\partial l(y_t, o_t)}{\partial w_h} = \frac{1}{T}\sum_{t=1}^{T} \frac{\partial l(y_t, o_t)}{\partial o_t}\frac{\partial g(h_t, w_o)}{\partial h_t}\frac{\partial h_t}{\partial w_h} \tag{1.25}$$

The last term $\frac{\partial h_t}{\partial w_h}$ is where the recursive dependencies of the model become significant. To determine $\frac{\partial h_t}{\partial w_h}$ one must take into account that h_t depends on both h_{t-1} and w_h, where computation of h_{t-1} also depends on w_h (Equation 1.26).

$$\frac{\partial h_t}{\partial w_h} = \frac{\partial f(x_t, h_{t-1}, w_h)}{\partial w_h} + \frac{\partial f(x_t, h_{t-1}, w_h)}{\partial h_{t-1}}\frac{\partial h_{t-1}}{\partial w_h} \tag{1.26}$$

Eliminating this recursion can be accomplished by writing the gradient computation as follows:

$$\frac{\partial h_t}{\partial w_h} = \frac{\partial f(x_t, h_{t-1}, w_h)}{\partial w_h} + \sum_{i=1}^{t-1}\left(\prod_{j=i+1}^{t}\frac{\partial f(x_j, h_{j-1}, w_h)}{\partial h_{j-1}}\right)\frac{\partial f(x_i, h_{i-1}, w_h)}{\partial w_h} \tag{1.27}$$

Now it can be seen that the number of products grows as the length of the input sequence increases. This leads to an outcome similar to the feedforward case where if a partial deriva-

tive component of that product is sufficiently large or small, the entire gradient calculation will "explode" or "vanish".

Long Short Term Memory

To address this shortcoming, the Long short term Memory (LSTM) model was proposed by Hochreiter and Schmidhuber [17]. In general, the advantage of this new LSTM approach is that an LSTM node is empowered to learn which past input signal characteristics are most relevant regardless of the extent to which those salient properties are separated in time from the current time step. This ability is facilitated by introducing a "gate" into a neural network node or cell [39]. Since the introduction of the LSTM cell by Hochreiter and Schmidhuber in 1997, many new variants have been proposed with the most widely implemented variant incorporating an additional gate termed the "forget gate" [11].

$$f_t = \sigma\left(W_{fh}h_{t-1} + W_{fx}x_t + b_f\right) \tag{1.28}$$

$$i_t = \sigma\left(W_{ih}h_{t-1} + W_{ix}x_t + b_i\right) \tag{1.29}$$

$$\tilde{c} = tanh\left(W_{\tilde{c}h}h_{t-1} + W_{\tilde{c}x}x_t + b_{\tilde{c}}\right) \tag{1.30}$$

$$c_t = f_t c_{t-1} + i_t \tilde{c}_t \tag{1.31}$$

$$o_t = \sigma\left(W_{oh}h_{t-1} + W_{ox}x_t + b_o\right) \tag{1.32}$$

$$h_t = o_t tanh(c_t) \tag{1.33}$$

The forget gate, $f(t)$, due to the sigmoid activation function is always a value between 0 and 1. This allows for $f(t)$ to control how much of the previous cell state $c(t-1)$ is maintained for subsequent calculations to determine the new hidden state $h(t)$ and cell state $c(t)$. The input gate, $i(t)$, controls what new information about the current input is included in the calculation of the new cell state $c(t)$. The cell state and the forget gate activations are the key to how LSTM nodes better avoid the exploding and vanishing gradient problems

of error backpropagation in RNNs. In the RNN models, using the chain rule ultimately led to a product of partial derivatives of $\frac{\partial h_j}{\partial h_{j-1}}$ (Equation 1.27). The motivation for LSTM was to make this recursive derivative have a constant value by introducing the cell state $c(t)$. However, this cell state added on its own would tend to grow without bound, thus the forget gate was introduced to scale the current cell state such that the update for the next time step did not grow uncontrollably. Much like the gradient for the RNN, the LSTM gradient depends on a recursive derivative for the cell state, $\frac{\partial c_j}{\partial c_{j-1}}$.

$$\begin{aligned}\frac{\partial c_t}{\partial c_{t-1}} &= \frac{\partial c_t}{\partial c_{t-1}} + \frac{\partial c_t}{\partial f_t}\frac{\partial f_t}{\partial h_{t-1}}\frac{\partial h_{t-1}}{\partial c_{t-1}} + ... \\ &... + \frac{\partial c_t}{\partial i_t}\frac{\partial i_t}{\partial h_{t-1}}\frac{\partial h_{t-1}}{\partial c_{t-1}} + \frac{\partial c_t}{\partial \tilde{c}_t}\frac{\partial \tilde{c}_t}{\partial h_{t-1}}\frac{\partial ht-1}{\partial c_{t-1}}\end{aligned} \tag{1.34}$$

The additive nature of the cell state partial derivative is distinct from that of the RNN hidden state partial derivative which only has a single term. This allows the LSTM to control the size of of this partial derivative by updating the constitutive gate components ($f(t)$, $i(t)$, and $\tilde{c}(t)$ such that summed partial derivatives of Equation 1.34 do not necessarily behave similarly making it less likely that all of the T gradients for BPTT will vanish or explode. In general, the LSTM is equipped to learn the gate values $f(t)$, $i(t)$, $\tilde{c}(t)$, and o_t such that the gradient is controlled. These features improve resistance to the gradient issues that arise with BPTT and make the LSTM the best ANN approach for learning data sequences.

1.2.3 Short term Synaptic Plasticity

Synapses are known to be dynamic on both long and short term time scales with the mechanisms driving short term synaptic dynamics demonstrated in this figure taken from a great review article on short term plasticity. The dynamics in general occur in three distinct phases of synaptic transmission. The first phase are dynamics controlling the availability of neurotransmitter filled vesicles to release transmitter into the synaptic cleft. The second phase includes dynamics that dictate how many of these vesicles are primed or ready to be released

upon the arrival of an axonal current. The last phase involves dynamics of neurotransmitter release and reuptake in the synaptic cleft. The set of these various dynamics varies across cell-type shaping how electrochemical signals are transmitted across synapses in the brain.

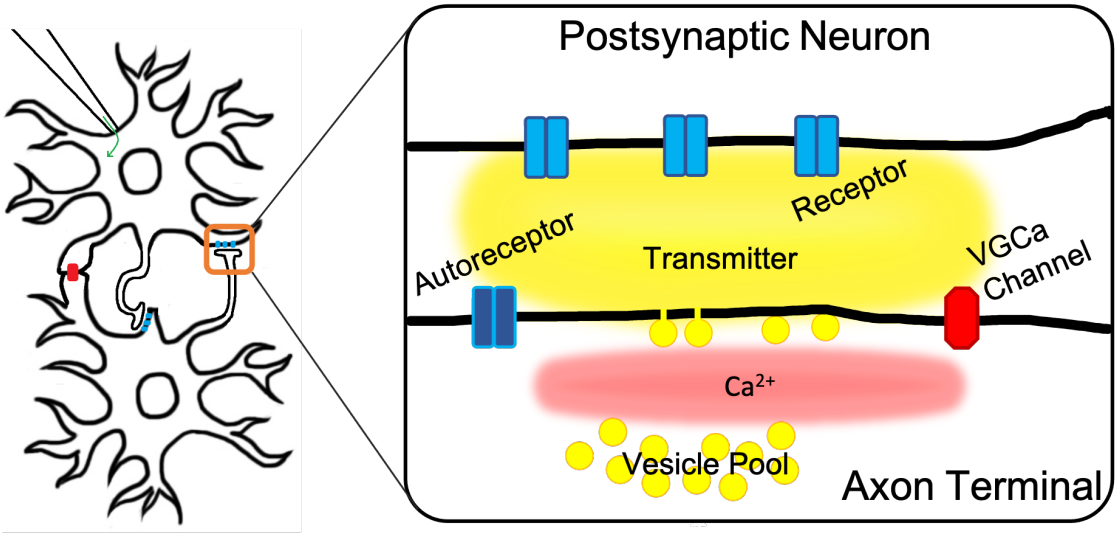

Figure 1.4: Short term synaptic plasticity is determined by biophysical processes involved in the transmission of presynaptic currents across the synaptic cleft to elicit postsynaptic potentials (PSPs) [16].

Experimental Evidence

In the past decade, electrophysiology experiments have demonstrated evidence of short term synaptic plasticity [43]. A recent example of such work sought to characterize cell type specific forms of short term synaptic plasticity in the adult mouse cortex [33]. While recording across a a subset of synaptic connections in mouse cortex, Seeman et al. injected stimulus trains consisting of 8 pulses, to induce STSP, followed by a variable delay period and then 4 more pulses to measure recovery. The 8 initial pulses allowed responses to reach a steady state, from which the extent of depression (or facilitation) could be observed at frequencies from 10 to 100 Hz. The 50 Hz stimulation protocol had additional recovery intervals ranging from 250 to 4000 ms. Depression of the elicited postsynaptic potential (PSP) was observed

across both the 8 initial pulses and the final 4 pulses. This demonstrated short term synaptic dynamics that depress synaptic strength over the course of presynaptic activity. Plotting the deconvolved PSP amplitudes at each input pulse normalized against the first PSP amplitude indicates frequency dependent depression in Sim1-sim1 connections (Figure 1.5A). Figure 1.5B demonstrates that this depression dynamic is observed for all cortical layers with the exception of L2/3. The spread of the amplitude ratios shown at left in Figure 1.5C shows that the L2/3 connections have a heterogeneity of dynamics with some connections depressing strongly (ratio < 1) and others facilitating (ratio > 1). These results show that not only is short term depression occurring in these cortical layers, but also that there is evidence for a diversity of synaptic plasticity mechanisms.

Mathematical Models

To better characterize the short term synaptic plasticity of the homogenous synapse classes studied in by Seeman et al. in [33], the PSPs were averaged over all available synapses depending on stimulation frequencies and delays between the 8th and 9th presynaptic pulses and models of synaptic dynamics were fit to the PSP data [25]. Comprehensive models of STSP were constructed by Lee et al. in [25] to involve dynamics of many different synaptic transmission factors. The workflow of the study began by using all the Seeman et al. PSP data gathered with 50 Hz stimulus sweeps. The PSP data was then clustered to find heterogeneity within the synapses of each cortical layer. Then thresholding was done by fitting a standard STSP model that has been previously demonstrated to explain empirical data well. Lastly, a set of different models, incorporating different synaptic transmission factors were fit to each cluster of PSP data to determine which mathematical descriptions best explain the data. If the classes of connections are indeed homogeneous the PSP data would be expected to optimally cluster, indicated by a minimum BIC value, in a single cluster. However, Lee et al. showed that for synapses in layers rorb, sim1, tlx3, and L2/3 the optimal cluster count was roughly 3 or 4 clusters [25]. This is evidence that even within cortical layers, there is a diversity of short term synaptic dynamics.

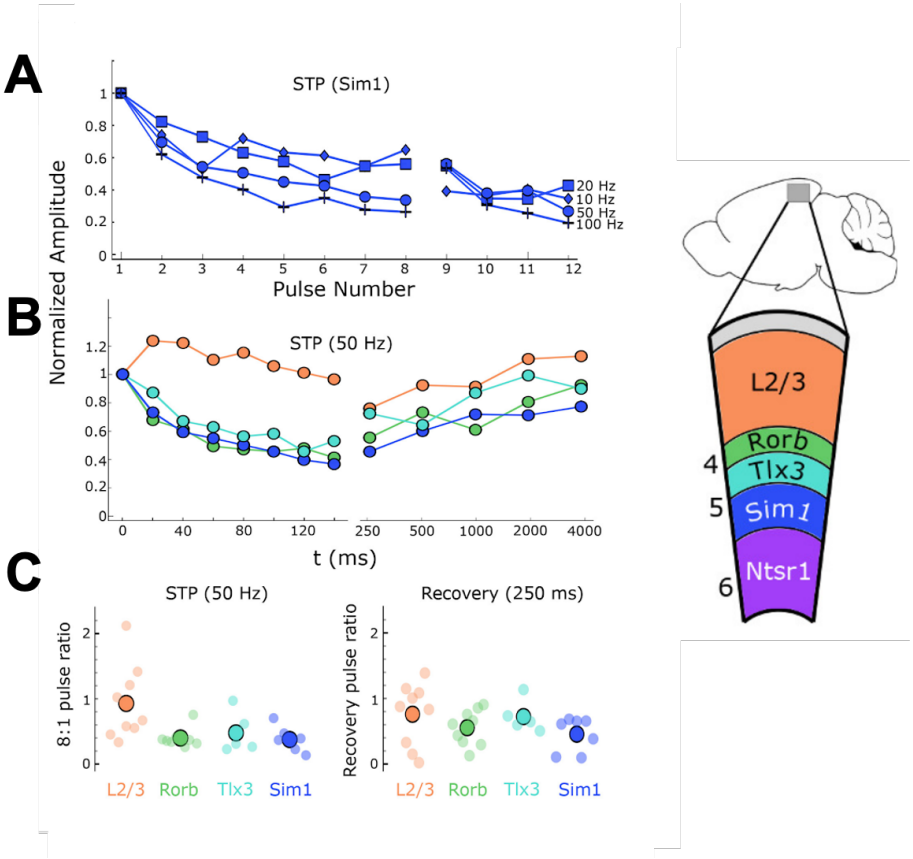

Figure 1.5: **A** Plot of PSP amplitudes at each input pulse frequency, normalized against the first PSP amplitude. **B** Normalized PSP amplitudes for synaptic connections in each layer for 50 Hz input pulse trains with 250 ms delay periods. **C** Left: Amplitude ratios of the last of the 8 initial pulses to the first. Right: Last to first pulse ratios for the recovery pulse trains following a 250 ms delay [33].

Having found strong evidence for heterogeneity of STSP, the Lee et al. study subsequently fit 12 different models of synaptic dynamics to each cluster of PSP data. These models were composite models of individual differential equation models, or "gates", describing distinct synaptic transmission biophysical dynamics (Figure 1.6). The first composite model gate was a depression, or vesicle depletion, model. This models presynaptic vesicles as a limited resource such that ongoing activity can lead to a suppression of the postsynaptic response via vesicle depletion. This process is described by a simple first order kinetic model where

$n(t)$ is the occupancy of the release pool of vesicles, bounded between 0 and 1, τ_r is the time constant of vesicle replenishment, and t_j is the presynaptic spike time. This model predicts an exponential decay of the postsynaptic response during stimulation at a constant rate, and an inverse relation between input frequency and steady state level of depression. Conversely, a facilitation mechanism can also underlie short term synaptic dynamics where an accumulation of residual calcium in the synaptic terminal causes rapid voltage gated calcium channel facilitation. As calcium concentration in the presynaptic terminal controls the release probability of the vesicles, a simple phenomenological model of a facilitation dynamic is to increase the release probability after each presynaptic spike. In the second model gate for this facilitation dynamic P_0 is the baseline release probability, and τ_f is the the recovery time constant (Figure 1.6). The third model gate, Use-Dependent Vesicle Replenishment, models the phenomena where vesicle replenishment can accelerate after intensive stimulation. This effect has been found to depend on an increase in intracellular calcium concentration, and to occur in a physiological range of input firing rates. One way to model this is by allowing presynaptic activity to directly modulate the time constant τ_r of vesicle replenishment. In this model each presynaptic action potential reduces the time constant by a factor α_{FDR}, the time constant τ_r recovers to its resting value τ_{r0} with a time constant τ_{FDR}. The desensitization model gate describes how the binding kinetics of the postsynaptic receptors change over the time course of signal transduction through the synaptic cleft. An approximation of the state of the population of receptors $S(t)$, can be modeled using first order kinetics. The quantity $S(t)$ represents the fraction of non-desensitized receptors. The rate of recovery from this desensitization is dictated by the time constant τ_D and the amount that presynaptic activity modulates the postsynpatic response is controlled by α_D. Lastly, the slow modulation of release model gate addresses the opposite case of use-dependent vesicle replenishment by modelling activity-dependent release probability suppression. Potential mechanisms behind this phenomena include voltage gated calcium channel inactivation or activation of presynaptic autoreceptors such as the metabotropic glutamate receptor or AMPA Receptors. This effect can be modelled by adding activity-dependent modulation of the baseline release prob-

ability P_0 within a composite model including depression and facilitation dynamics. In the slow modulation of release model gate the baseline release probability $P_0(t)$ is reduced by a constant fraction α_i after each spike, and recovers back to $\tilde{P}_0$ with a time constant τ_i.

These individual model gates are combined to form 12 different composite models which were subsequently fit to the Seeman et al. PSP data to determine which dynamics best describe the STSP for the unique synapse clusters. Lee et al. found that individual or small subsets of these composite models uniquely fit different synapse clusters thus further characterizing the heterogeneity of biophysical processes controlling the observed short term synaptic dynamics within layers of the cortex. This is motivation for studying STSP further to understand the role it plays in network computation.

Temporal Dynamics

Depression (DEP) $\frac{dn}{dt} = \frac{1-n}{\tau_r} - P \cdot n \cdot \delta(t - t_k)$

Facilitation (FAC) $\frac{dP}{dt} = \frac{P_0 - P}{\tau_f} - P_0(1-P)\delta(t - t_k)$

Use-dependent Replenishment (USR) $\frac{d\tau_r}{dt} = \frac{\tau_{r0} - \tau_r}{\tau_{FDR}} - \alpha_{FDR} \cdot \tau_{FDR} \cdot \delta(t - t_k)$

Desensitization (DSR) $\frac{dS}{dt} = \frac{1-S}{\tau_D} - \alpha_D \cdot n \cdot P \cdot S \cdot \delta(t - t_k)$

Slow Modulation of Release (SMR) $\frac{dP_0}{dt} = \frac{\tilde{P}_0 - P_0}{\tau_i} - \alpha_i \cdot P_0 \cdot \delta(t - t_k)$

Models

Model number	Temporal Dynamics	Model number	Temporal Dynamics
1	DEP	7	DEP, FAC, DSR
2	DEP, FAC	8	DEP, FAC, USR, DSR
3	DEP, USR	9	DEP, FAC, SMR
4	DEP, DSR	10	DEP, FAC, USR, SMR
5	DEP, USR, DSR	11	DEP, FAC, DSR, SMR
6	DEP, FAC, USR	12	DEP, FAC, USR, DSR, SMR

Figure 1.6: In the study conducted by Lee et al. composite models of varying complexities were formulated using combinations of individual temporal dynamic models [25].

Short Term Synaptic Plasticity in ANNs

A recent study that sought to implement a short term synaptic plasticity model in an ANN architecture used a depression only model of STSP. The ANNs were trained to perform a "change detection task" where the network indicates when a sequence of images switches to

a new image. These models were trained alongside animal models performing the same task. Image data was passed in to three distinct model types all with a single hidden layer of 16 neurons. The ANN models were (1) a standard RNN, (2) a feedforward model with adapting synapses connecting the input to hidden layer, and (3) a RNN with half of the synapses from the input to hidden layer having short term synaptic dynamics. The adapting synapses had connection strengths that depended on vesicles modelled as a limiting resource, much like the depression model in Figure 1.6. The results from training these models demonstrate that the The RNN model with STSP input to hidden layer connections trains much faster than the vanilla RNN and feedforward STSP models (Figure 1.7A). A second key finding was that the models with adapting synapses showed a small degree of response matrix asymmetry similar to the animal model task performance, while the RNN model instead performed the task with a high degree of matrix symmetry (Figure 1.7B). This suggests that not only does STSP impact learning speed in ANNs, it also pushes the ANNs towards more biological response characteristics.

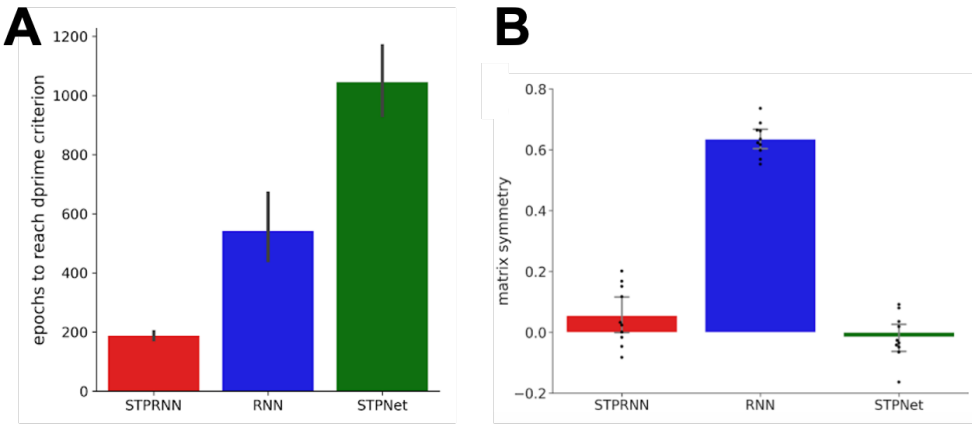

Figure 1.7: **A** D-prime, or the sensitivity index, is estimated by the Hit rate, or the correct change detection, minus the false alarm rate. Plotted here is th number of epochs to reach the early stopping D-prime criteria for each model. **B** Response probability matrices were constructed for change detection transitions between different image sequences. Plotted here are measures of response probability matrix symmetry bounded between –1 and 1, with negative values indicating asymmetry and positive values indicating symmetry [18].

1.3 Project Description

In ANNs, updating the connection weights during training represents a mechanism akin to long-term plasticity [12]. However, neurons also exhibit strong, and strongly diverse, short term dynamics – and these are missing from almost all ANN models. These short term synaptic plasticity mechanisms (STSP) can be broadly categorized as short term depression and short term facilitation. Recent work has shown how STSP can provide networks with short term memory [28] and the ability to easily solve detection of change tasks [18]. In addition, recent work in ANNs has incorporated maintenance of temporary state information by implementing "fast weights" in a standard RNN layer architecture [1]. The authors accomplish this by constructing a "fast" weight matrix in parallel with the standard weight matrix. The fast weight matrix depends on the correlation of current and past hidden states, and its output contributes additively to the evolution of the network hidden state at each time step. Our approach differs in several interesting ways. Chief among them is that the (fast) synaptic dynamics we study represent a multiplicative adaptation at the level of all synapses emerging from an active neuron, rather than a synapse-specific facilitation of synaptic strength for *pairs* of co-active synapses, as in [1]. A second difference is the method of training the parameters for the short term plasticity, which in our model is based on the same optimization as the synaptic weights.

Here, we build on these results in three major ways. First, we allow the timescale of STSP in each individual neuron to be learned, simultaneously with standard learning of overall synaptic weights, via unified gradient-based tools to minimize task training loss in Pytorch (extending from Hu et al.)[18]. Second, we draw explicit comparison to standard RNN and LSTM models with related, but distinct, mechanisms. And third, we study the role of STSP in prediction, a fundamental temporal computation. Prediction tasks are inspired by predictive coding, which has a long tradition in signal processing [8] and has been studied for more than two decades in computational neuroscience [30, 19] as well as more recent work on prediction in the machine learning and ANN literature (e.g. with recurrent convolutional

LTSMs [27] and video prediction). While predictive coding training functions have been shown to reproduce several features observed in neuronal data [30], there are also differences [42]. Here, we explore the role of STSP in a very simplified dynamical system prediction task as well as a more complex sequential MNIST prediction task.

Very recent work has reached similar scientific conclusions to ours, namely that heterogeneous intrinsic dynamics help temporal integration [29] but there are key differences which make these studies complementary. First, we focus on STSP dynamics, as opposed to cellular and synaptic integration time constants. Second, we focus on temporal prediction rather than classification. Finally, we work with rate-based networks, which enables us to make comparison with standard RNN and LSTM machine learning tools.

Chapter 2

RESULTS

2.1 Task description

We investigated the role of STSP in learning by training a set of neural network models with varying usages of STSP dynamics to predict future sequences in two different task environments: time series prediction of N-dimensional sinusoidal and N-dimensional decaying exponential dynamics (Figure 2.1A), and prediction of pixel intensity in sequentially read MNIST images (Figure 2.1B).

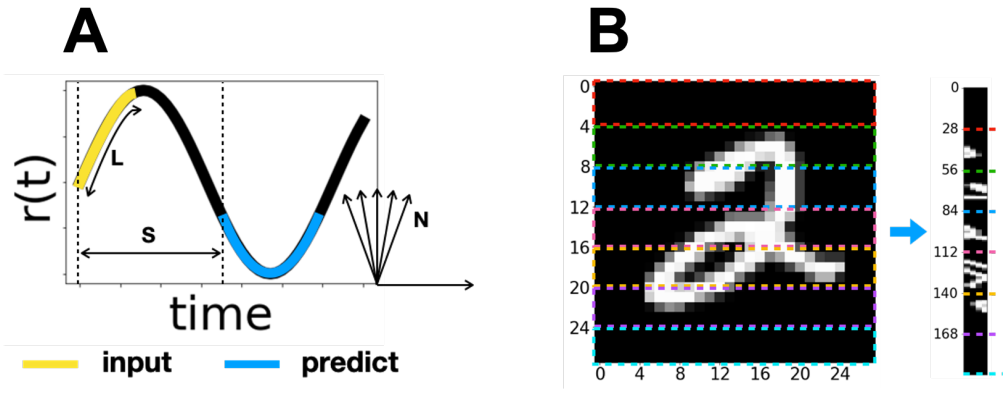

Figure 2.1: **A** Models were trained to perform a time series prediction task on an N-dimensional dynamic. The number of steps ahead to predict is S and the input sequence length is L. **B** An example transformation of an MNIST image into an MNIST pixel sequence. Four row sections of the original image are horizontally stacked to generate a 4 dimensional sequence.

2.2 Model description

We implement a model for short-term depression [6], similar to Hu et al., in firing-rate network models. The rate of change of synaptic resources for neuron i with short-term depression synapses is given by Equation 2.1, where $x_i(t)$ represents the synaptic resources, U is a constant, $r_i(t)$ represents the presynaptic activity at time t, and τ_{x_i} is the time constant of the synapses formed by neuron i. The firing rate of a given postsynaptic neuron j is represented by Equation 2.2. Here, W_{ij} represents the overall synaptic strengths subject to long-term plasticity as in standard ANN learning, and subject to additional modulation on faster timescales via the STSP rule [18]. The non-linearity, Φ, is a ReLU activation function.

$$\frac{dx_i(t)}{dt} = \frac{(1 - x_i(t))}{\tau_{x_i}} - U x_i(t) r_i(t) \qquad (2.1) \qquad r_j(t + dt) = \Phi\left(\sum_{i=0}^{N} W_{ij} x_i(t) r_i(t)\right) \qquad (2.2)$$

All models have 3-layer architectures with a hidden layer of varying size to facilitate parameter matching across models (Figure 2.2). Parameter matching was done by fixing the hidden layer dimension at 16 for the RNN and matching the other models to the corresponding trainable parameter count by modulating the hidden layer dimensions (Appendix A.0.1). Short-term synaptic dynamics were included in feedforward models (STSP, STSPtau, STSPdiv) connecting the input layer to hidden layer, as well as in RNNs (RNSTSP, RNSTSPtau) with the STSP synapses within the hidden layer (Figure 2.2). These models with short-term plasticity are differentiated in both the initialization of the short-term dynamic time constants and the dynamics of these time constants across training. In the STSP and RNSTSP models, the time constants for each hidden node are homogeneous ($\tau_x = 3$) and static. The STSPtau and RNSTSPtau models likewise have all time constants initialized homogeneously ($\tau_x = 1$), but the *time constants are learned model parameters* trained via the gradient of the task loss, thus permitting the values to be optimized across training. The time constants for the STSPdiv model, by comparison, are static parameters but are initialized at heterogeneous values based on the final learned time constants for the STSPtau model

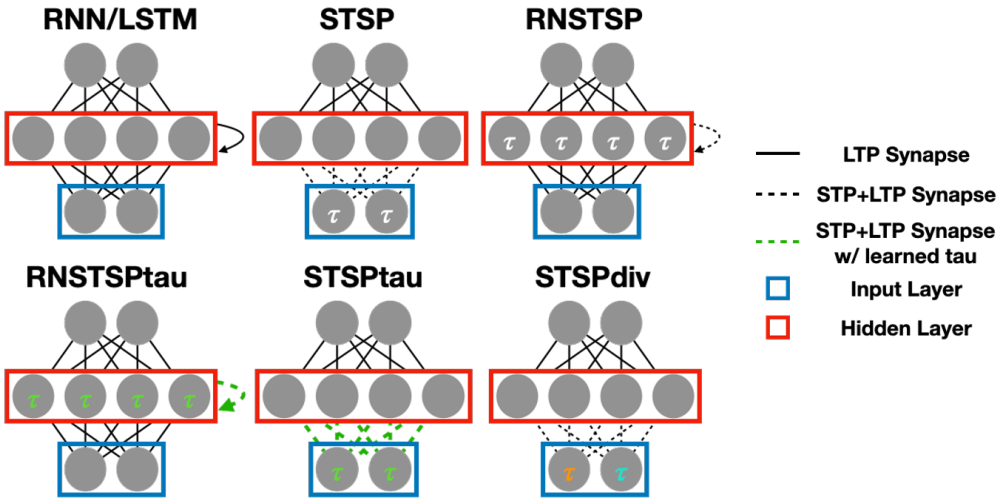

Figure 2.2: The network models. White parameters are static and homogeneous. Green parameters denote time constants that are learned. Multicolored parameters are static but heterogeneous. STP denotes short-term plasticity synapses and LTP indicates long-term plasticity connections, as in standard ANN learning.

2.3 STSP improves learning of temporal dynamics

2.3.1 Parameter Matched Models

The results presented here use sinusoidal input stimuli of dimensions 4, 8 and 16 that have frequencies along each dimension which are evenly spaced across a range of frequencies from 0.001 to 0.333 Hz (Table 2.1). Our results indicate that a significant performance advantage is conferred to the RNNs with STSP. This benefit is evidenced by lower minimum loss values for the RNSTSP and RNSTSPtau models across the training and test sets (Figure 2.3). A Kruskal-Wallis test shows significant ($P < .001$) differences in the final 200 epochs of the validation loss between the standard RNN and LSTM models and the RNSTSP, and RNSTSPtau models. A post hoc Mann-Whitney test shows significant differences in the validation loss across the final 200 epochs, for both dimension 8 and 16 sinusoidal inputs, between the following model pairs: RNSTSPtau and RNN ($P < .001$), RNSTSP and RNN ($P < .001$), RNSTSPtau and LSTM ($P < .001$), and RNSTSP and LSTM ($P < .001$).

Table 2.1: Distribution statistics for the N-dimensional sinusoid input dynamic frequencies when the values for each dimension are selected to evenly span the range from 0.001 to 0.333 Hz.

Dimension	**Mean**	**Median**	**Std. Dev.**	**Range**
4	0.070	0.038	0.078	0.195
8	0.060	0.032	0.064	0.195
16	0.042	0.014	0.056	0.195

Furthermore, training the time constants in the RNSTSPtau and STSPtau models results in observable diversification (Figure 2.4,2.5). In addition, possessing a diversified, or diversifiable, set of STSP dynamics differentiates model performance in feedforward model architecture. This differentiation is evidenced by the STSPtau and STSPdiv models significantly outperforming the STSP model (Figure 2.3). A Kruskal-Wallis test shows significant ($P < .001$) differences in the final 200 epochs of the validation loss between the STSP, STSPtau, and STSPdiv models. A post hoc Mann-Whitney test shows significant differences between the model pairs of STSPtau and STSP ($P < .001$), and STSPdiv and STSP ($P < .001$) for dimension 8 and 16 sinusoidal inputs.

The models were also trained to predict 4, 8, and 16 dimensional decaying exponential dynamics. The most notable finding from training the models on these dynamics is that within the first 200 epochs of training, the models with STSP are achieving lower training and testing loss more quickly than the vanilla LSTM and RNN models (Figure 2.6). As expected most models perform this task reasonably well, but for the higher dimensional case (Figure 2.10C) there is qualitative evidence that the feedforward architectures with STSP and trainable or fixed heterogeneous time constants (STSPtau, STSPdiv) not only learn the fastest, but also may perform the best after 1000 epochs. In addition, similar to the sinusoidal dynamic prediction, a performance difference is observed between the STSP model with static, homogenous time constants, and the STSPtau and STSPdiv model variants with learnable or fixed time constant heterogeneity. The overall higher loss values across

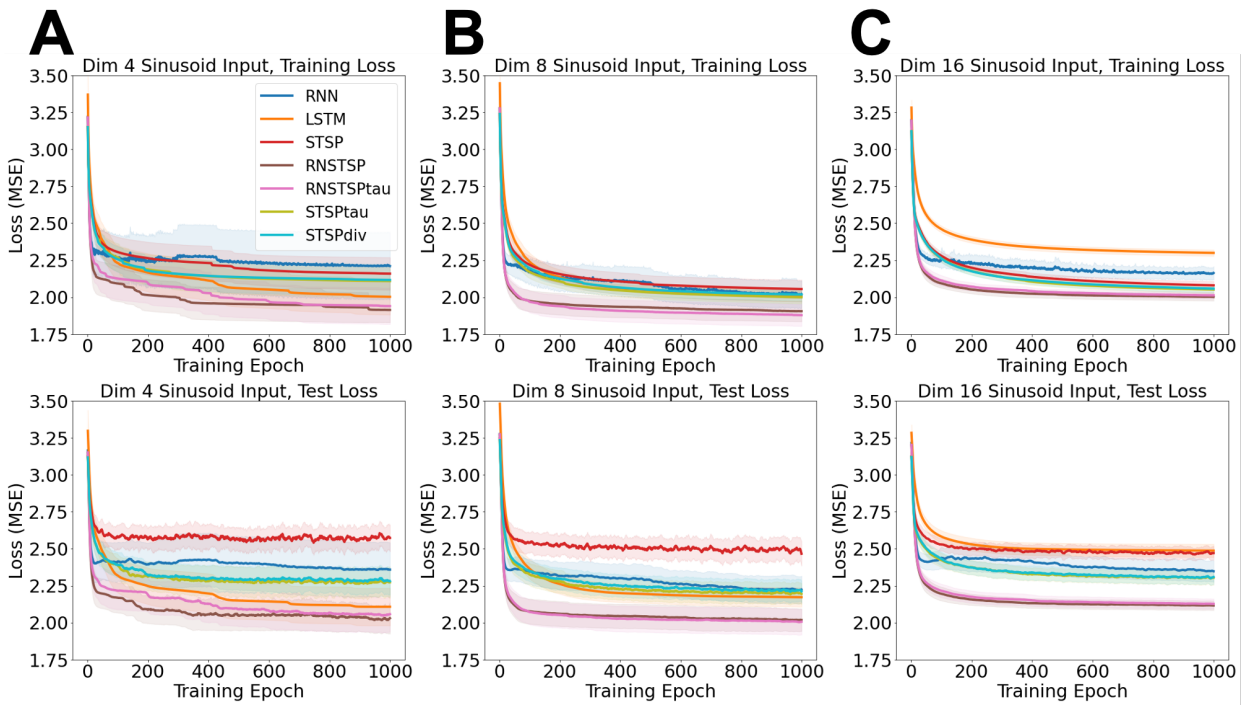

Figure 2.3: Training (top) and test (bottom) loss for parameter matched models predicting **A** 4, **B** 8, and **C** 16 dimensional sinusoid inputs.

training and testing in the exponential decay prediction is attributable to the normalization of the mean squared error loss (Equation 2.5.1) by the input signal variance (Figure 2.6). A decaying exponential dynamic of the same amplitude and duration as a sinusoidal dynamic as has much lower variance and therefore the variance normalized loss is larger for these models though the dynamic should be easier to predict. In addition to the model performance, similar to the sinusoidal dynamic prediction task, the time constants for the RNSTSPtau and STSPtau models become heterogeneous across training (Figures 2.7, 2.8). However the final learned distributions of the time constants have smaller mean and variance. The dynamics of the RNSTSPtau time constants across training for the exponential decay prediction are differentiated from those of the sinusoidal prediction task in that the trajectories approach a steady state instead of continual growth. The dynamics of the STSPtau time constants for

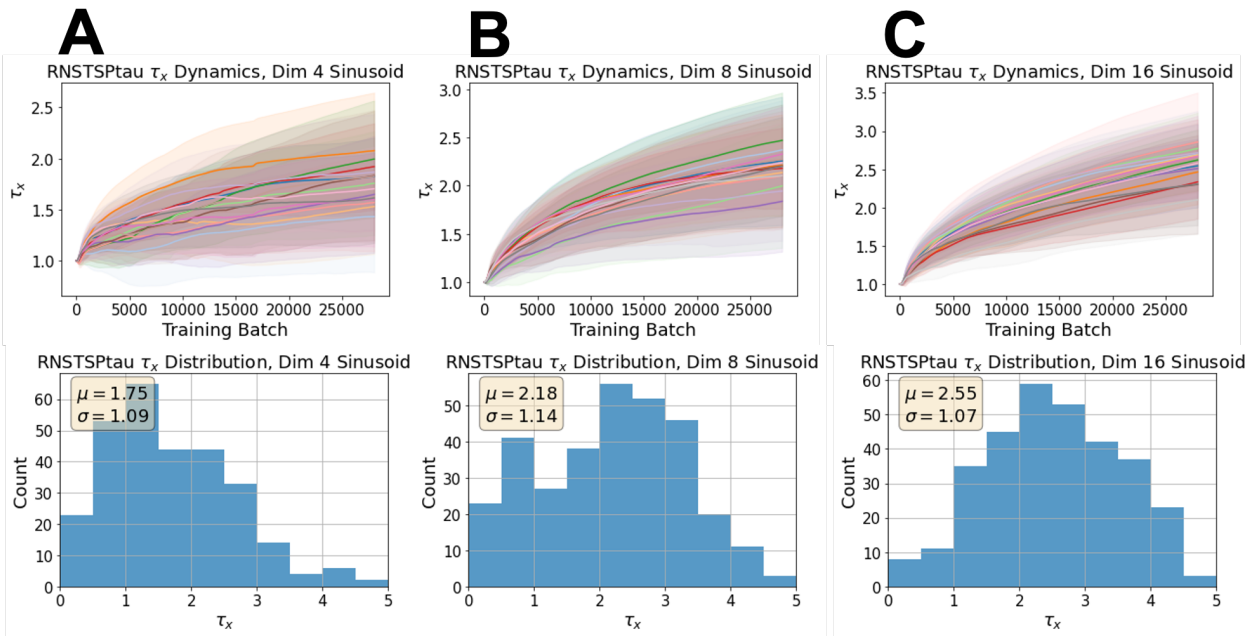

Figure 2.4: Dynamics (top) and final trained distributions (bottom) of STSP time constants for the parameter matched RNSTSPtau models predicting **A** 4, **B** 8, and **C** 16 dimensional sinusoid inputs.

both prediction tasks appear to approach steady state values. Interestingly, the STSPtau model average learned time constants are markedly different from those of the RNSTSPtau model in either prediction task suggesting that neural network architecture may dictate what kind of short term synaptic plasticity is useful. Outside of these observations, the relationship between the time constant dynamics across training and neural network computation remains to be explored.

2.3.2 Hidden Layer Matched Models

As a control for hidden layer dimension versus trainable parameter count, models were trained on the same sinusoid and exponential decay prediction tasks with matched hidden layer dimensions of 16 nodes. The trainable parameter count for each model is reported in

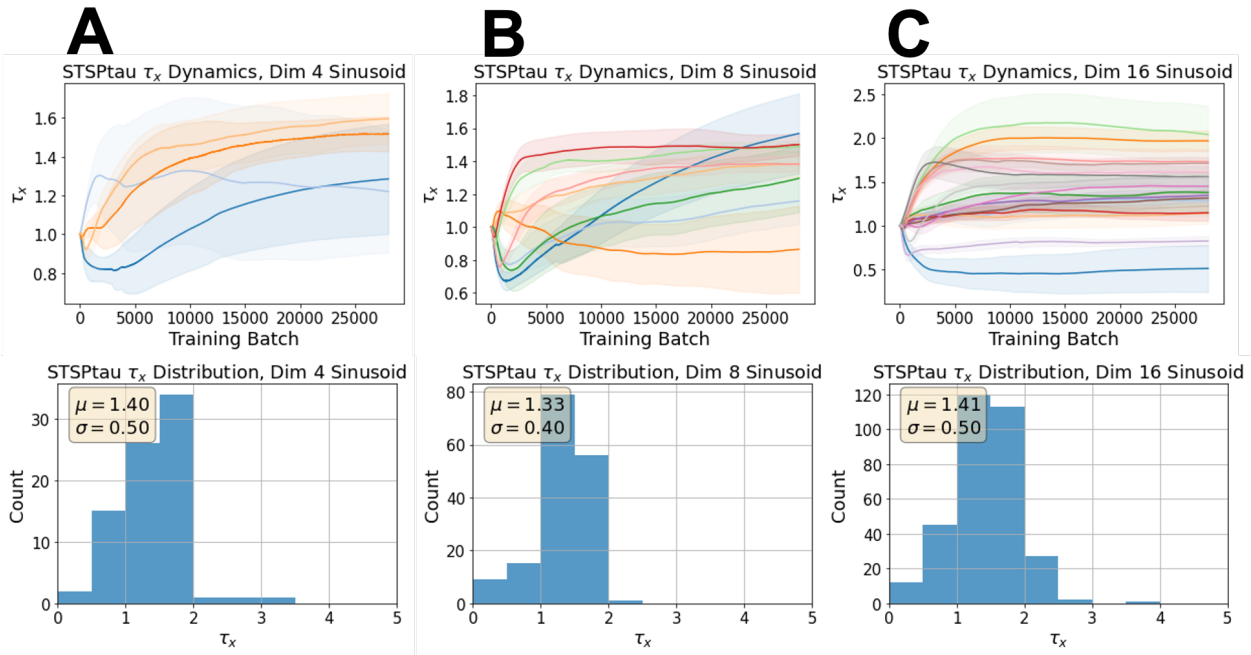

Figure 2.5: Dynamics (top) and final trained distributions (bottom) of STSP time constants for the parameter matched STSPtau models predicting **A** 4, **B** 8, and **C** 16 dimensional sinusoidal inputs.

Table A.4 within Appendix A.0.1. Equipped with more trainable parameters, the LSTM model now demonstrates superior performance attaining the lowest training and test loss on 4, 8 and 16 dimension sinusoidal prediction (Figure 2.9). However, the RNSTSP and RNSTSPtau models attain comparable final test loss values and in the 4 and 8 dimensional sinusoidal prediction cases, these models learn faster than the LSTM (Figure 2.9A, B). On the exponential decay prediction, model performance is largely comparable across task dimensionalities (Figure 2.10).

2.3.3 Large Hidden Layer Models

To test a large network paradigm the dimension of the hidden layer for each model was expanded to 256 nodes. After 1000 training epochs, the LSTM model achieves the lowest

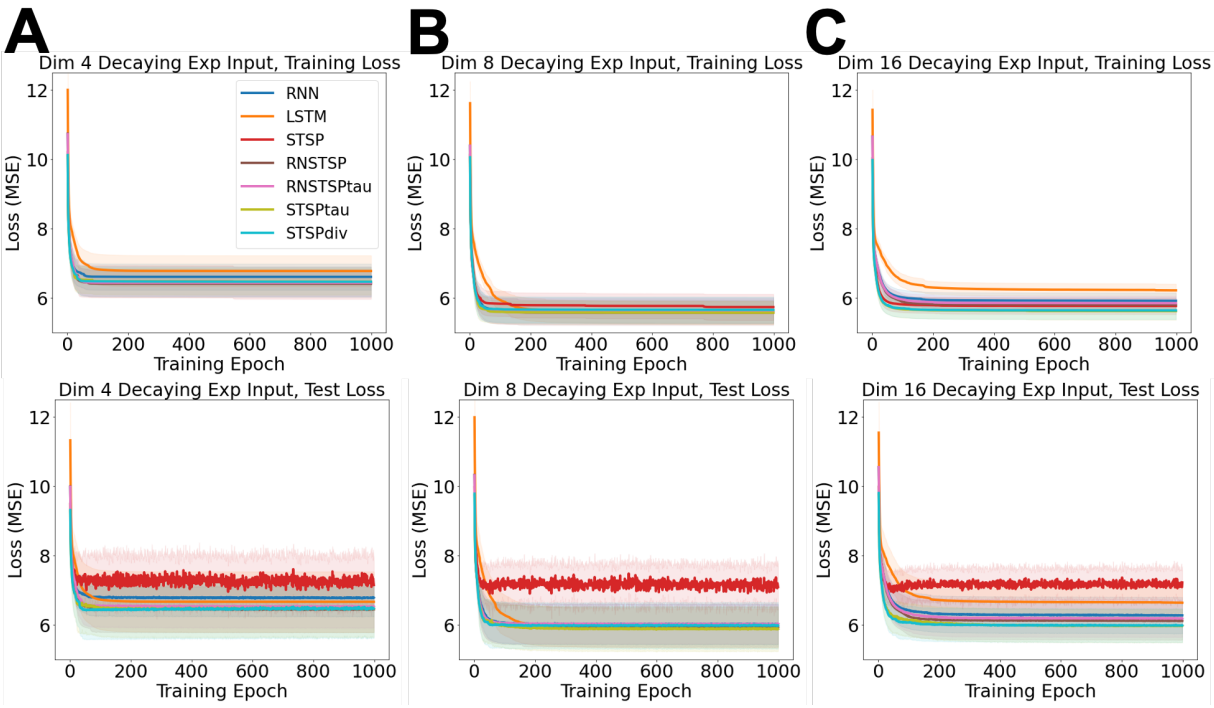

Figure 2.6: Training (top) and test (bottom) loss for parameter matched models predicting **A** 4, **B** 8, and **C** 16 dimensional decaying exponential inputs.

test loss of all the models for the 4 and 8 dimensional sinusoidal prediction tasks, but does not significantly outperform the RNSTSP and RNSTSPtau models for the 16 dimension sinusoidal prediction (Figure 2.11). Considering that the LSTM uses significantly more trainable parameters in these test cases, the efficiency of the models with STSP is evident, especially as the dimensionality of the prediction task is increased. Overfitting is observed for all models with 256 hidden nodes as evidenced by test loss increasing across training (Figure 2.11). The RNSTSP and RNSTSPtau models achieve overall minimum validation loss values comparable to the LSTM model. The RNSTSP and RNSTSPtau models achieve this minimum validation loss earlier than the LSTM model in the case of predicting the 4 dimensional sinusoid. When the number of units is much larger than task dimensionality, the RNSTSP and RNSTSPtau models appear especially prone to overfitting. Therefore adopting

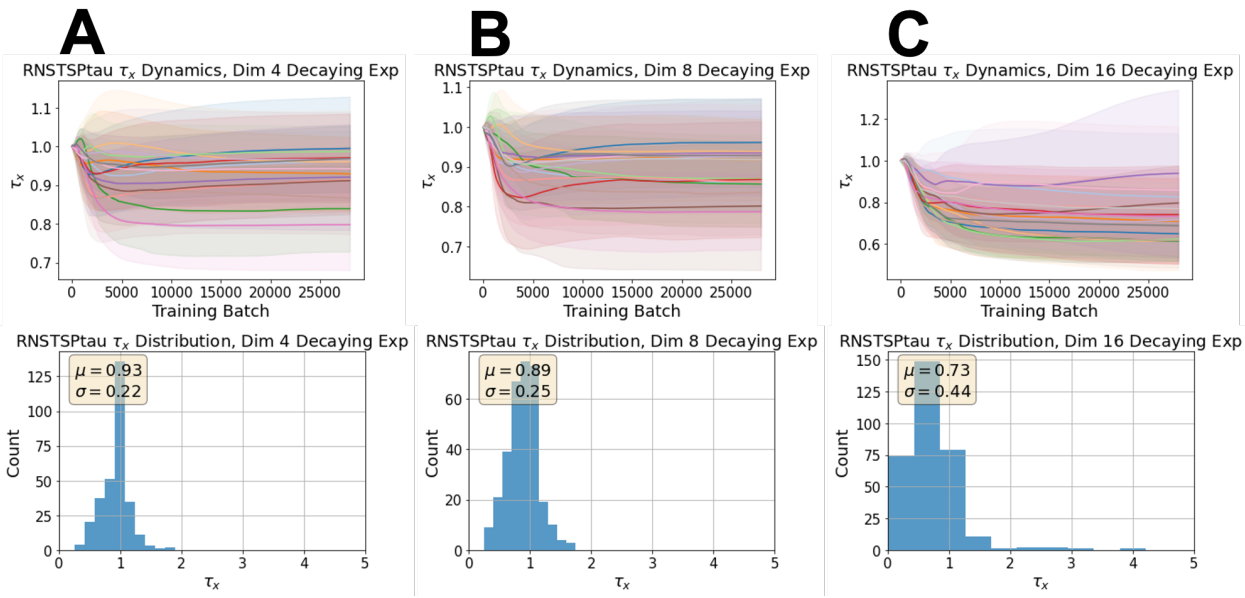

Figure 2.7: Dynamics (top) and final trained distributions (bottom) of STSP time constants for the RNSTSPtau models predicting **A** 4, **B** 8, and **C** 16 dimensional decaying exponential inputs.

an early stopping criteria during training is necessary to allow these models to continue to perform well in large network implementations.

2.3.4 Random Input Sinusoid Frequencies

In contrast to the stimuli for all model performance results shown previously, the results illustrated in this section are from models trained to predict a N-dimensional sinusoidal dynamic where each dimension frequency is selected randomly from a uniform distribution spanning 0.001 to 0.333 Hz (Table 2.2). Previously, the frequencies for each dimension of the input sinusoid were explicitly selected to represent frequencies evenly spaced across a range from 0.001 to 0.333 Hz (Table 2.1). This instance of sinusoidal stimuli is thus referred to as "diverse" in the sense that it contains heterogeneous frequencies that cover the range of allowable frequencies. In comparing the statistics of the sinusoidal input frequency content

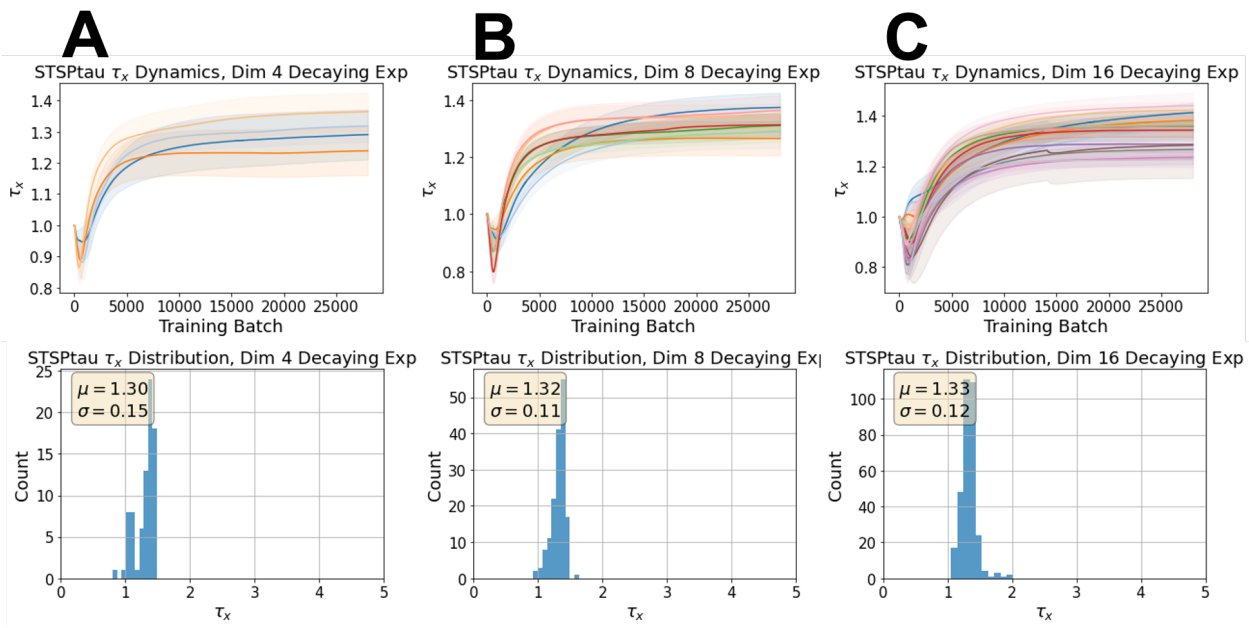

Figure 2.8: Dynamics (top) and final trained distributions (bottom) of STSP time constants for the STSPtau models predicting **A** 4, **B** 8, and **C** 16 dimensional decaying exponential inputs.

between the diverse and this random design strategy, the range is now uncontrolled, and generally smaller, especially in the 4 dimensional sinusoid case. For dimensions 8 and 16, the overall ranges are comparable to the diverse sinusoid signal case, as are the standard deviations, but the averages are notably larger. Therefore, we would generally expect the same results as seen previously in Figure 2.3B, and C, but in the 4 dimensional case we might expect the models to perform more comparably as the input signal is less diverse due to a smaller range and standard deviation of frequencies (Table 2.2 vs Table 2.1). In general our expectations are met, in that in Figure 2.12B, and C we see the models differentiated exactly as they were in Figure 2.3B, and C with RNSTSP and RNSTSPtau as the best performing models. However across task dimensions, the minimum test loss of ~ 2.25 is larger than the minimum test loss of ~ 2.00 for the diverse input sinusoid case. This indicates that this task

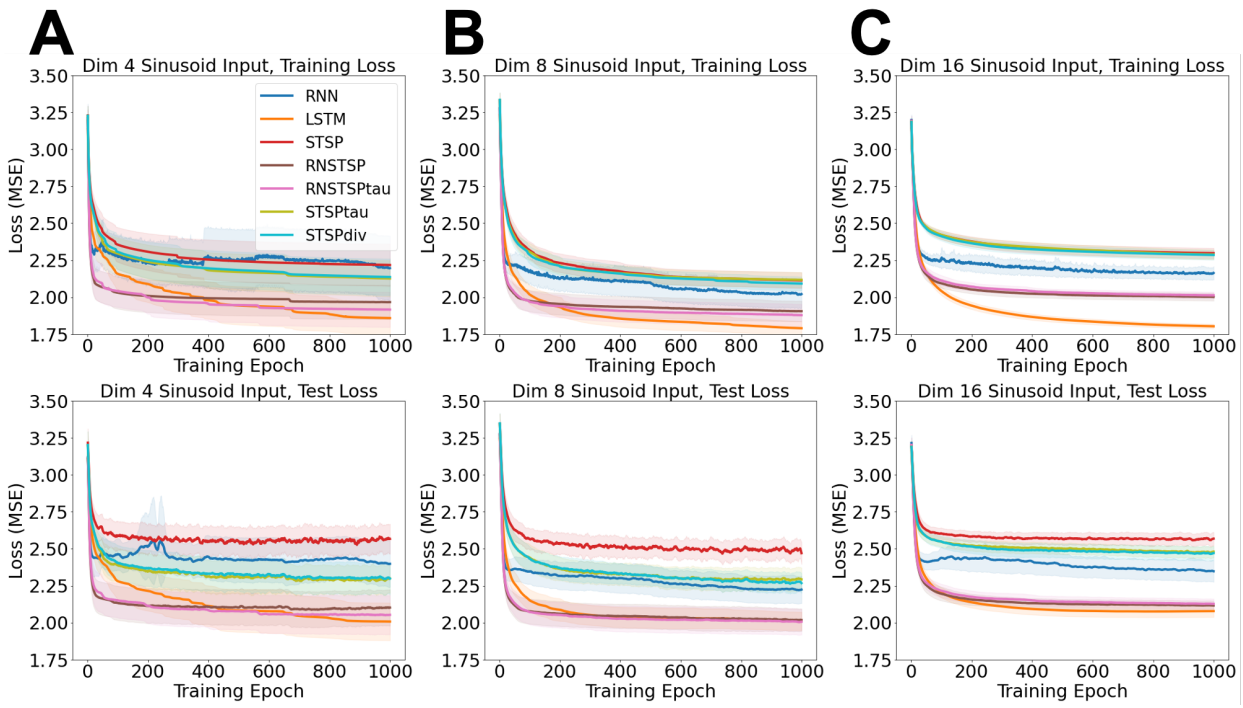

Figure 2.9: Training (top) and test (bottom) loss for hidden layer dimension matched models predicting **A** 4, **B** 8, and **C** 16 dimensional sinusoidal inputs.

is more challenging for the models. Perhaps the prediction difficulty for an oscillatory signal does not depend simply on the breadth of the frequency content. What is markedly different between the diverse frequency values used before and these randomly selected values is that the the median frequency values are much larger indicating that the signal frequency content is skewed to higher frequencies. From a curve fitting perspective, it may be intuitive why it is more challenging to predict a N-dimensional sinusoid with on average higher frequency content. Modelling higher frequency signals as a linear combination of polynomials requires a higher order polynomial, and consequently a less parsimonious model, to accurately represent the signal.

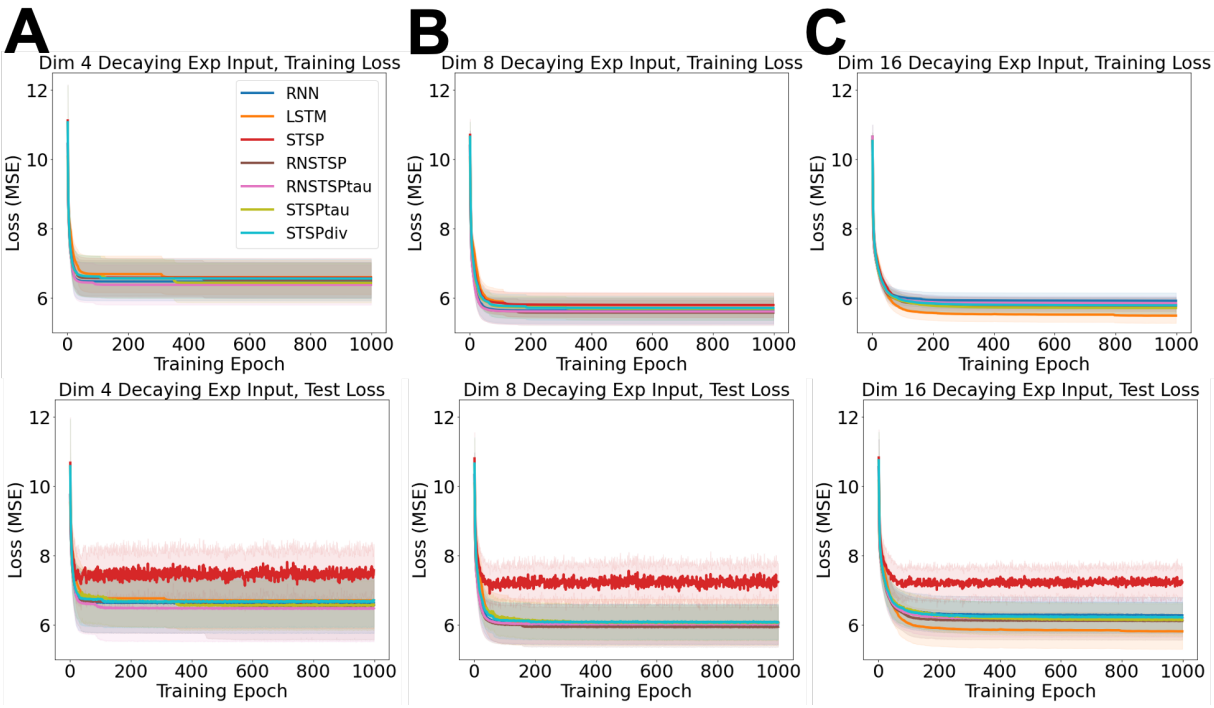

Figure 2.10: Training (top) and test (bottom) loss for hidden layer dimension matched models predicting **A** 4, **B** 8, and **C** 16 dimensional decaying exponential inputs.

2.3.5 Sliding Window Cross Validation

All previous models were validated using a random assignment cross validation scheme described in Section 2.5.1. This results in training batches containing input sequences that may not be contiguous in time within the overall dynamic to be predicted. Though this scheme ensures that the input sequences are not biased towards inlcusion in the train or test set, it may not be the most biologically realistic approach. Most likely real world dynamics are learned in a continuous manner where as more data comes in to the system new associations are learned. One can imagine however that perhaps the input signal is first fed through a subnetwork that randomly orders temporal sections of the input to improve learning performance in the downstream prediction network. Nonetheless, it is important to also train our

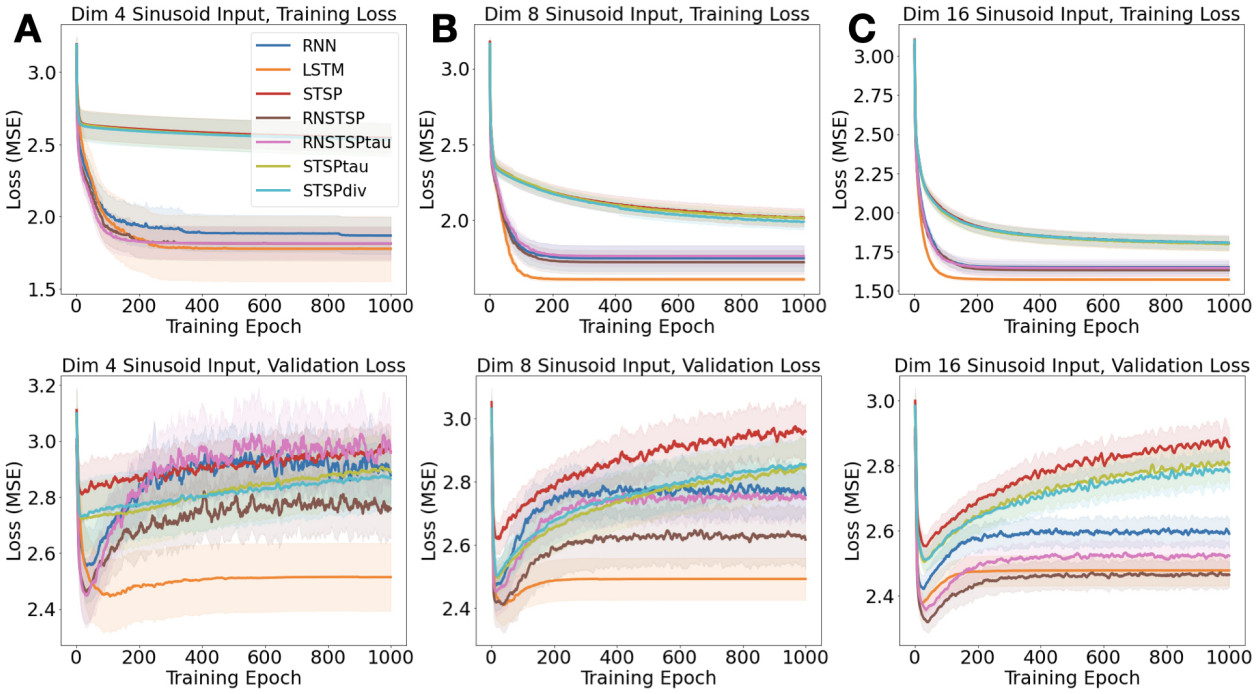

Figure 2.11: Training and validation loss for models, each with 256 hidden layer nodes, predicting (**A**) 4, (**B**) 8, and (**C**) 16 dimensional sinusoids. Unlike Figure 2.3, models here are matched for the number of units, resulting in more parameters for LSTM (Table A.5). The frequencies along each dimension were again selected to be evenly spaced across a range of frequencies from 0.001 to 0.333 Hz.

models with a more biologically plausible, and in general more commonly implemented, cross validation scheme for time series data: the sliding window method. Briefly, this method divides the input signal into sections, equal to the number of desired validation folds, that are continuous in time. These sections are then divided into smaller continuous sections for the train and test sets. This ensures that in either data set the input sequences fed through the models are always next to one another in the time series. The main result from employing this strategy is that all models fail to generalize as well as was previously observed using random assignment cross validation. This is evidenced by the lower bound on the test loss being $\sim 3.5 - 4.5$ for all models across sinusoid dimensionality as compared to ~ 2.0 for

Table 2.2: Distribution statistics for the N-dimensional sinusoid input dynamic frequencies when the values for each dimension are selected randomly from a uniform distribution.

Dimension	**Mean**	**Median**	**Std. Dev.**	**Range**
4	0.051	0.059	0.033	0.079
8	0.073	0.074	0.054	0.149
16	0.084	0.077	0.059	0.187

models trained with the random assignment strategy (Figure 2.13). This makes sense as we would expect randomly assigning input sequences would ensure that neither the training or testing sets are biased towards including input sequences from a particular time period of the time series. This perhaps could be remedied by overlapping the the cross validation sections instead of reserving all data in a given validation fold for that particular fold of training and testing. In general models trained with this cross validation scheme perform worse and the performance differences previously demonstrated between RNN models with STSP and the vanilla RNN and LSTM models (Figure 2.3) is not observed in this scenario.

2.4 STSP improves learning of MNIST pixel sequences

We also compared the performance of our models on a sequential MNIST prediction task. The models were trained on a subset of the full Lecun et al. MNIST dataset [24] containing only the digits 2 and 8 to create a simple but not trivial task. The images were preprocessed such that the pixel row sequences were stacked along 4 input dimensions. The models were trained to predict 4 steps ahead along the pixel sequence.

Our results again indicate that a significant performance advantage is conferred to the RNNs with STSP (Figure 2.16B, C). A Kruskal-Wallis test shows significant ($P < .01$) differences in the final 200 epochs of the validation loss between the standard LSTM model and the RNSTSP, and RNSTSPtau models. A post hoc Mann-Whitney test shows significant differences in the final 200 epoch validation loss between the RNSTSPtau and LSTM ($P <$

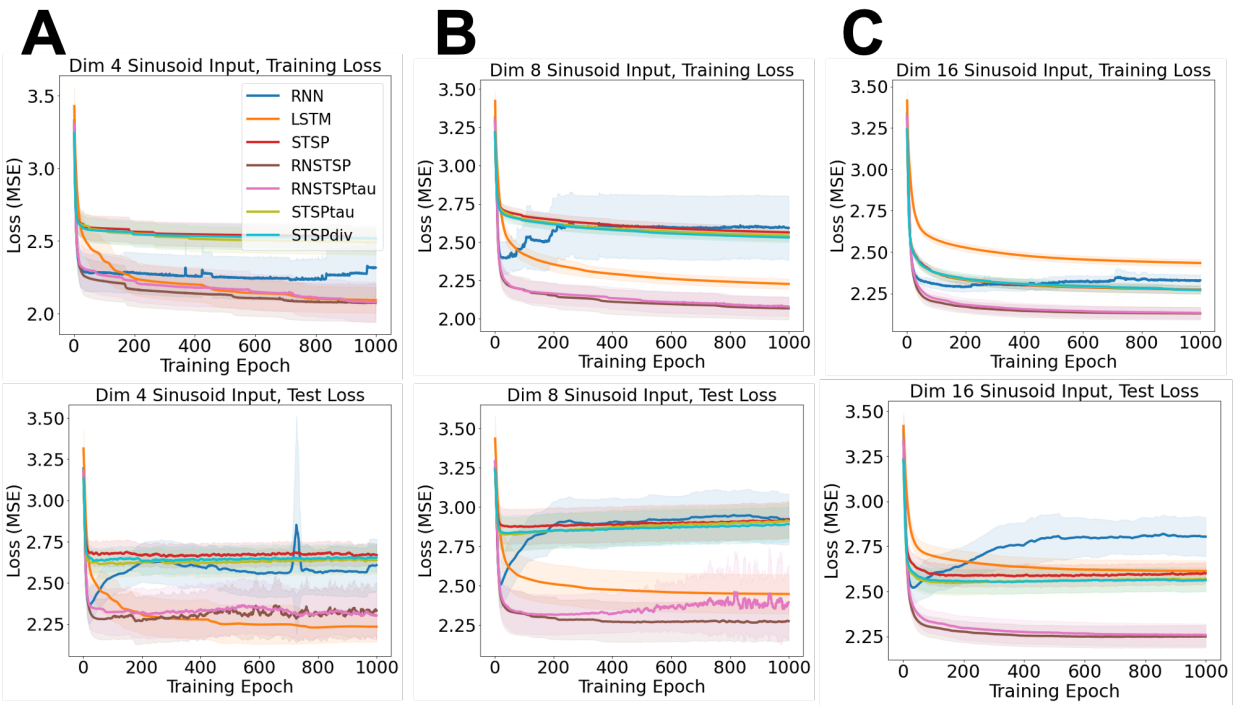

Figure 2.12: Training (top) and test (bottom) loss for hidden layer dimension matched models predicting **A** 4, **B** 8, and **C** 16 dimensional sinusoidal inputs with frequencies along each dimension selected randomly.

.001), RNSTSP and LSTM ($P = .008$), and RNSTSPtau and RNN ($P = .053$) model pairs. No statistically significant performance differences are observed due to heterogeneity of STSP dynamics.

2.5 Methods

2.5.1 Model Implementation

For both tasks the models were trained using the ADAM optimizer with the default settings (learning rate of 0.001 and beta values of 0.9, 0.999). Models were trained for 1000 epochs with 20 cross-validation folds using mean squared error loss (Equation 2.5.1). For the N-dimensional sinusoid and exponential decay prediction tasks, to assign the data to the

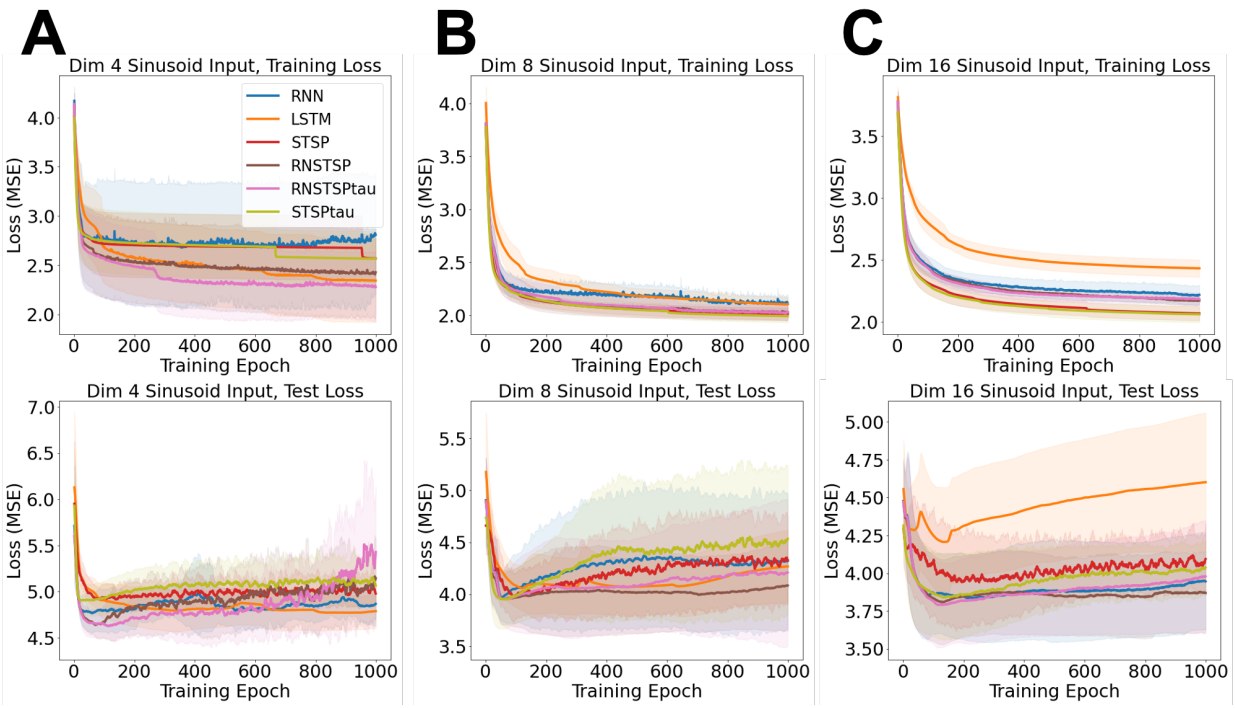

Figure 2.13: Training (top) and test (bottom) loss for parameter matched models predicting **A** 4, **B** 8, and **C** 16 dimensional sinusoidal inputs using sliding window cross validation.

train, test, or validation set, the time series was first divided into segments of the desired number of sequential time steps (i.e. sequence length, L) and then these segments were randomly assigned (without replacement) to either the training, validation, or testing set in the proportions 70%, 15%, and 15%. The input sequences in each data set were then batched together based on the desired batch size. An individual batch represents the amount of data provided to a model per forward pass. A second cross validation scheme using a sliding window was also implemented with results detailed in Section 2.3.5. In contrast to the random assignment cross validation approach, the sliding window approach divides the time series to predict into contiguous sections for each validation fold. Within each of these sections the first 70% of the data is used for training and the final, 30% for testing. Notably, with the sinusoidal inputs, the desired number of cross validation folds determines the lowest frequency

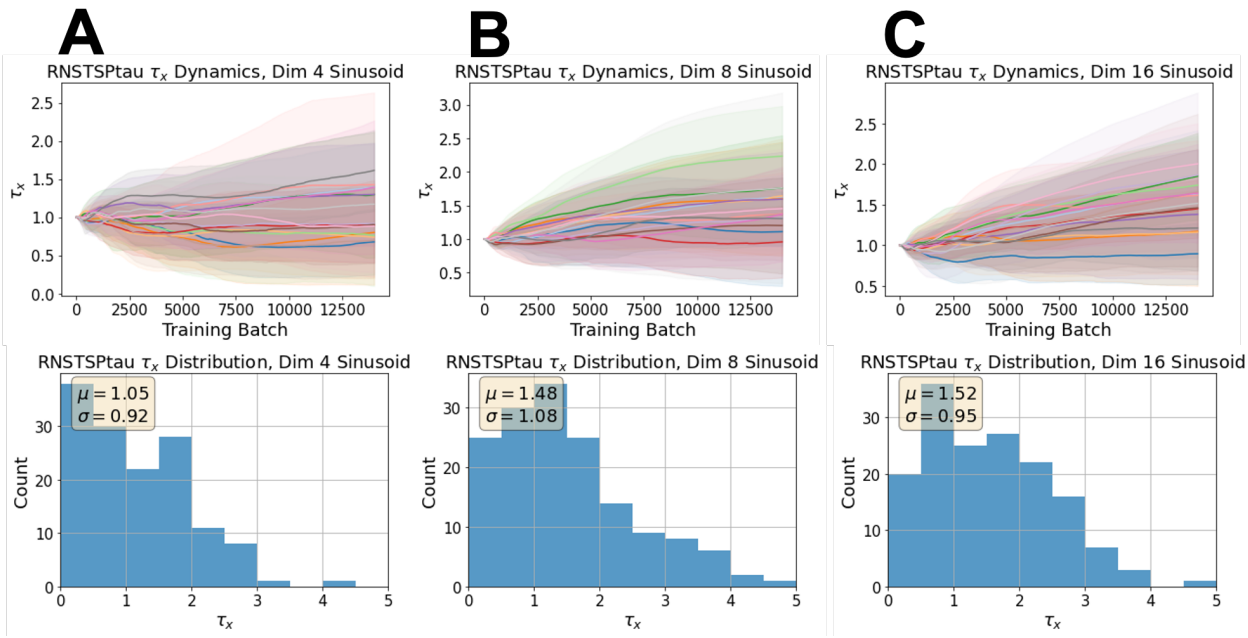

Figure 2.14: Dynamics (top) and final trained distributions (bottom) of STSP time constants for the RNSTSPtau models predicting **A** 4, **B** 8, and **C** 16 dimensional sinusoidal inputs using sliding window cross validation.

that the sinusoidal input signal can contain. This is because if a given validation fold does not at least contain a single complete cycle of the sinusoid, the model will not be able to sufficiently learn the dynamic in any of the validation folds.

Of note, for presentation purposes the test loss plots for all figures are smoothed using a moving average with a window size of 3 epochs. Additionally, due to exploding gradients, only 18 CV folds were used for the RNN model predicting the dimension 16 sinusoid in Figure 2.3C. Similarly, in Figure 2.16, only 15, 19, and 19 CV folds were used for the RNN, RNSTSP, and RNSTSPtau models respectively.

$$Loss = \frac{1}{n}\sum_{i=0}^{n}\left(\frac{y_i - \hat{y}_i}{\sigma_{in}}\right) \tag{2.3}$$

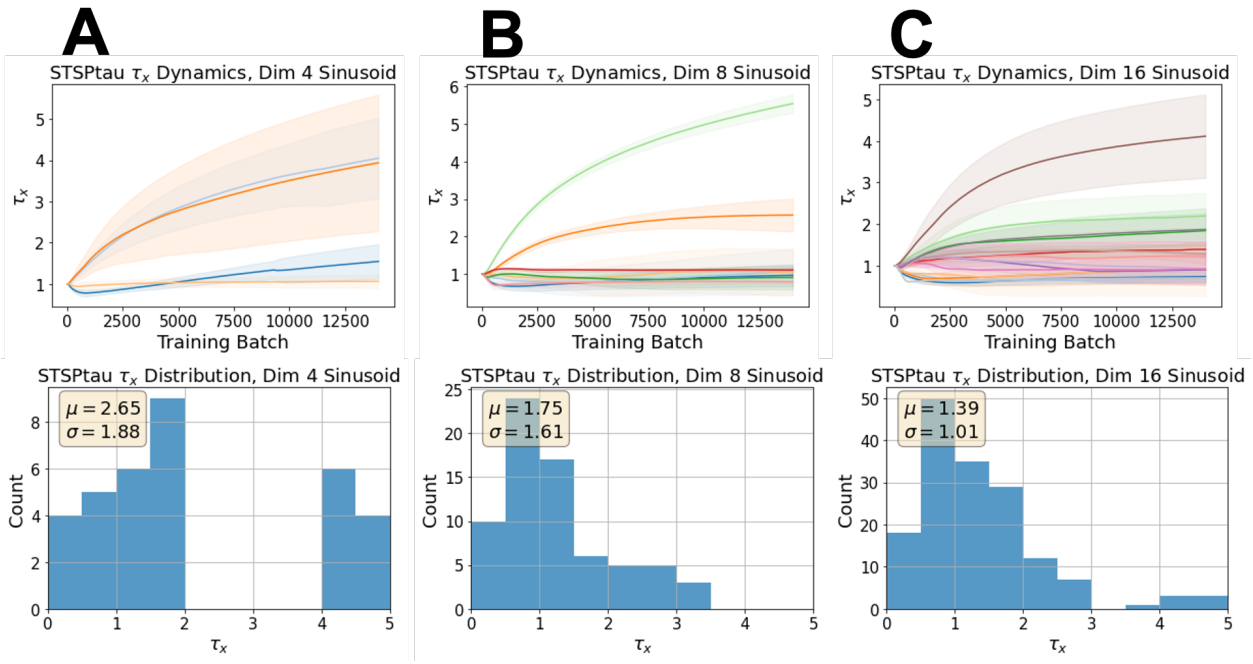

Figure 2.15: Dynamics (top) and final trained distributions (bottom) of STSP time constants for the STSPtau models predicting **A** 4, **B** 8, and **C** 16 dimensional sinusoidal inputs using sliding window cross validation.

2.5.2 Python Library

A python library was developed to implement these models in the context of both the sinusoid and decaying exponential dynamic prediction and the sequential MNIST prediction tasks. Beginning with the models, each is defined as a Pytorch module and can be parameterized by input, hidden, and output layer dimension as well as the number of hidden layers. For models including STSP synapses, biophysical parameters are also included as inputs and modulate the time constant initialization, the synaptic constant U (See Equation 2.1), and the percentage of synapses to assign with STSP (See Appendix B).

For training a model on the sinusoidal or exponential decay prediction tasks one can

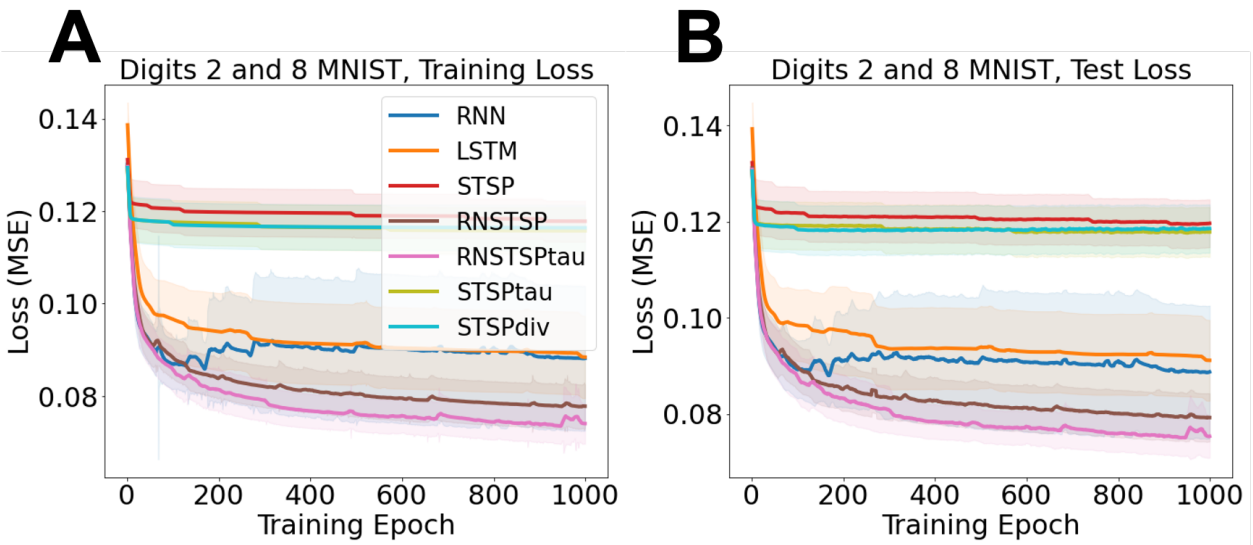

Figure 2.16: **A** An example transformation of an MNIST image into an MNIST pixel sequence. Four row sections of the original image are horizontally stacked to generate a 4 dimensional sequence. **B** Training and **C** validation loss for predicting 4 steps ahead along the MNIST sequence.

make a function call to "runParameterSet" in the utils.py library. This function appears as:

```
def runParameterSet(dim, noise_std, steps, random_seed, model_type,
                    input_data_type,cv_folds=1, input_length=10, hidden_dim=16,
                    batch_size=5, n_layers=1, epochs=1000, diverse_stim = False,
                        **kwargs):
```

This function itself makes calls to stand alone sinusoid/exponential decay data generators and a train and test function to return a dictionary data structure containing epoch loss, batch loss, test output, test set accuracy by epoch, and test set accuracy by batch. The function takes in as parameters:

- dim: value indicating the dimension of the input data
- noise_std: standard deviation for noise distribution

- steps: the number of steps ahead to predict at each point in time
- random_seed: random seed
- model_type: model object 'type' can be: STSP, RNN, LSTM, RNN_STSP, RNN_STSP_tau, RNSTSP, or RNSTSP_tau
- input_data_type: type of input data either 'sinusoid' or 'decaying_exponential'
- cv_folds: number of cross validations to perform
- input_length: length of input sequence within the training batches. Default 10.
- hidden_dim: dimension of hidden layer of desired model type. Default 16.
- batch_size: mini-batch size. This is the amount of input sequences the model will see between backpropagation updates. Default 5.
- n_layers: number of hidden layers for desired model type. Default 1.
- epochs: number of training epochs. An epoch is demarcated as the training batches required to pass all training data through the model one time. Default 1000.
- *kwargs: key word arguments, in this case the accepted key word arguments are:
  - model_seed: can be used to specify a intialization seed for the model
  - STSP_perc: can be used to specify how many connections (percentage) with have synaptic dynamics
  - tau_init: optional initilization time constant distribution specification. Options are: rand_uniform, rand_normal, fixed_dist. See models.py for more information
  - tau_x: optional specification of the time constant value(s).
  - optimizer_type: can be SGD or Adam. Default Adam.
  - hyperparams: optional dictionary containing hyperparameter values for SGD or Adam. can specify as many or as few as desired. Default is to use default values according to Pytorch documentation.

This set of parameters gives the user control over practically every parameterization of the models themselves and the training process. Sample code is included in Appendix B.

The entire codebase including utility functions and models will be made available to the community through Github.

2.6 A Mechanistic Hypothesis on the Role of STSP

Synaptic plasticity in a linear node facilitates embedding of the time series thus expanding the order of the filter that the node is able to produce. Consider the same system as Figure 1.2, but with a synaptic resource variable $x(t)$ with intrinsic dynamics described by first order kinetics (Equation 2.4)

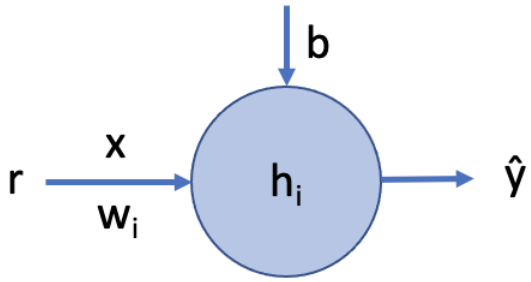

Figure 2.17: Single linear node with synaptic resource variable.

$$\frac{dx}{dt} = \frac{1 - x(t)}{\tau_x} - Ux(t)r(t) \tag{2.4}$$

$$\text{Backward Euler: } x(t) = x(t-1) + \frac{1 - x(t-1)}{\tau_x} - Ux(t-1)r(t) \tag{2.5}$$

$$\hat{y}(t) = \sum_{i=0}^{q=0} w_i r(t) x(t) + b \tag{2.6}$$

$$\hat{y}(t) = \sum_{i=0}^{k} w_i r(t) \left(x(t-1) + \frac{1 - x(t-1)}{\tau_{x_i}} - Ux(t-1)r(t) \right) + b \tag{2.7}$$

If we assume a single neuron model like Figure 2.17 with a hyperbolic tangent activation function:

$$\text{Let: } h(t) = w_i r(t) \left(x(t-1) + \frac{1 - x(t-1)}{\tau_{x_i}} - U x(t-1) r(t) \right) + b \tag{2.8}$$

$$\hat{y}(t) = \frac{2h(t) + \frac{(2h(t))^2}{2!} + \frac{(2h(t))^3}{3!} + \frac{(2h(t))^4}{4!}}{2 + 2h(t) + \frac{(2h(t))^2}{2!} + \frac{(2h(t))^3}{3!} + \frac{(2h(t))^4}{4!}} \tag{2.9}$$

Expanding this to single layer, multi-node feedforward and reccurent networks yields:

$$\text{Feedforward: } h(t) = \left[\mathbf{W}^{hx} r(t) \left(x(t-1) + \frac{1 - x(t-1)}{\tau_{x_i}} - U x(t-1) r(t) \right) + \mathbf{b}^h \right] \tag{2.10}$$

$$\text{RNN: } h(t) = \left[\mathbf{W}^{hh} h(t-1) + \mathbf{W}^{hx} r(t) \left(x(t-1) + \frac{1 - x(t-1)}{\tau_{x_i}} - U x(t-1) r(t) \right) + \mathbf{b}^h \right] \tag{2.11}$$

One can observe that including these STSP dynamics would lead to different Volterra kernel approximations for the individual nodes within the networks (Section 1.2.1). These approximations are differentiated from the vanilla feedforward and RNN versions in that for a given time step, they include encodings of the previous input signal time step via $x(t-1)$. Moreover, a second set of parameters, the τ_{x_i}'s, can be optimized over in addition to the weights w_i. These observed differences leads to the hypothesis that STSP meaningfully modifies the adaptive polynomial filters that a single neuron, or network of neurons, individually and/or collectively approximate. Future work will be directed towards analytically representing these differences further and estimating coefficient values from single node and network simulations. Collectively these results may demonstrate the impact that STSP has on neural network computation.

Chapter 3

DISCUSSION

3.1 Conclusions

Our results in the first tasks studied, that of predicting a simple dynamical system, demonstrate a performance advantage for RNSTSP (RNNs with short-term synaptic dynamics) over standard RNNs and LSTMs. This holds across a set of predicted dimensions (Figure 2.9) and dynamics (Figure 2.10) when the number of computational units is of the same order of magnitude as the task dimensionality. These results are confirmed in the second, more complex task, that of predicting sequential MNIST pixels (Figure 2.16). When the number of computational units significantly exceeds the task dimensionality, the RNSTSP models learn rapidly (Figure 2.11) but quickly start to overfit. A potential explanation for RNSTSP performance is the availability of the activity history in the short-term synaptic efficacy. We can view the short-term plasticity as a biological way to implement long short-term memory.

Our second finding is that heterogeneity in the time constants of STSP dynamics, though emergent in both recurrent and feedforward models with learned time constants (Figures 2.4 and 2.5), significantly improves learning only in the feedforward setting (Figure 2.9). Recurrent architectures may already allow nodes to manufacture different time scales of activity through the arbitrarily long or short activity trajectories within the recurrent layer. This property could make "ready-made" synaptic timescales, via STSP, a less significant resource for predicting time varying signals. In the feedforward setting, while diversity in these time constants was important, we did not find evidence that they needed to be learned on a neuron-specific basis in tandem with connection weights. Specifically, we found identical performance for the STSPtau model, in which this tandem learning did occur, and the STSPdiv variant, in which the heterogeneous time constants were pre-assigned and held fixed during

learning (Figure 2.9 and Figure 2.16). This indicates that effective learning of these time constants in nature could be in response to developmental or learning pressures on larger time scales than individual stimulus and task learning. A related idea is that once time constants have been learned to solve a given task, they will likely be of service in solving other tasks with relevant information over similar time scales. This said, the situations in which heterogeneity of synaptic dynamics impacts learning remain to be thoroughly explored. We believe that our findings open doors to future work along these lines, which should analyze temporal tasks which are gradually more complex and more ethologically relevant. To complement this, future efforts should also be made to demonstrate the theoretical underpinnings of how synapses acting as diverse dynamical components can facilitate the representation – and, more importantly, prediction – of time varying inputs.

3.2 Future Directions

Many new questions stem from the results of this thesis. With respect to the current results: How does time scale of STSP relate to time scale/complexity of input? We have seen that overall test loss varies depending on the frequency content (Section 2.3.4), but the relationship between the STSP time constant(s) and the time scales of the predicted signal remain to be explained. This relationship undoubtedly depends on where the STSP is included in the architecture. Therefore a promising place to begin may be with simple feedforward architectures with STSP synapses. Similarly, the impact on prediction of the relationship between time scale of STSP and the time scale of standard ANN synapses plasticity (LTP) should be characterized.

Aside from these questions relating to the parameterization of the models, a much broader set of open questions relate to: How does STSP (both homogeneous and heterogeneous) impact neural network computation involved in prediction? To answer this, approaches to understanding the nonlinear dynamics of the networks need to be explored. Previous work has been successful in detailing how the fixed point structure facilitates RNN computation on certain tasks and these results offer motivation for seeking to understand the impact of

STSP on ANNs in prediction tasks through dynamical systems analysis tools [**?**].

3.2.1 Nonlinear system identification

One approach to conceptualizing the dynamics of ANNs with STSP during time series prediction is to look for periodic orbits in the state space of the ANNs during prediction. The hypothesis is that though these orbits may exist even for vanilla RNN models, the presence of STSP synapses would expand network's ability to map input sequences to periodic trajectories in state space. A source of motivation for this future direction is the idea of Poincaré maps. Briefly, Poincaré maps, or first return map, is where a periodic orbit in the state space of a continuous dynamical system intersects with a specifc n-1 subspace, called the Poincaré section, Σ, which is transverse to the flow of the system. If one considers a periodic orbit with initial condition x_0 within a section of the space V, which leaves that section afterwards, and observes the point at which this orbit first returns to the section, x_1 then a map can be created to send the first point x_0 to the second x_1 which is the first recurrence map or Poincaré map. Perhaps something analogous is learned by ANNs to predict time varying signals such as the N-dimensional sinusoid we have presented results for previously. A possible path forwards to characterizing this behavior, would perhaps be to find non-linear oscillators that fit the hidden state activities of the ANNs during training and subsequently characterize the periodic orbits of those oscillators.

3.2.2 Hidden Layer Decoder

Another avenue to understanding how these ANNs are performing the computations associated with prediction would be to train a second readout layer to decode from the state space of the hidden layer the frequency, in the case of sinusoidal prediction, or the numerical digit classification in the case of the sequential MNIST prediction task. The performance of this decoder would shed light on whether the dynamics in the hidden layer are classifiable and thus, perhaps to predict the input dynamics, the networks are also performing a classification of discrete dynamic states.

BIBLIOGRAPHY

[1] Jimmy Ba, Geoffrey E Hinton, Volodymyr Mnih, Joel Z Leibo, and Catalin Ionescu. Using fast weights to attend to the recent past. In D. Lee, M. Sugiyama, U. Luxburg, I. Guyon, and R. Garnett, editors, *Advances in Neural Information Processing Systems*, volume 29. Curran Associates, Inc., 2016.

[2] A. D Back and A. C Tsoi. Fir and iir synapses, a new neural network architecture for time series modeling. *Neural computation*, 3(3):375–385, 1991.

[3] Y Bengio, P Simard, and P Frasconi. Learning long-term dependencies with gradient descent is difficult. *IEEE transactions on neural networks*, 5(2):157–166, 1994.

[4] Steven L. (Steven Lee) Brunton. *Data-driven science and engineering : machine learning, dynamical systems, and control*. Cambridge University Press, Cambridge, United Kingdom ; New York, NY, 2019.

[5] Vincenzo Capasso and David Bakstein. *An Introduction to Continuous-Time Stochastic Processes: Theory, Models, and Applications to Finance, Biology, and Medicine*. Modeling and simulation in science, engineering and technology. Springer, New York, NY, 2015.

[6] Peter Dayan and L. F. Abbott. *Theoretical Neuroscience: Computational and Mathematical Modeling of Neural Systems*. Massachusetts Institute of Technology Press, Cambridge, Mass., 2001.

[7] Saskia E J de Vries, Jerome A Lecoq, Michael A Buice, Peter A Groblewski, Gabriel K Ocker, Michael Oliver, David Feng, Nicholas Cain, Peter Ledochowitsch, Daniel Millman, Kate Roll, Marina Garrett, Tom Keenan, Leonard Kuan, Stefan Mihalas, Shawn Olsen, Carol Thompson, Wayne Wakeman, Jack Waters, Derric Williams, Chris Barber, Nathan Berbesque, Brandon Blanchard, Nicholas Bowles, Shiella D Caldejon, Linzy Casal, Andrew Cho, Sissy Cross, Chinh Dang, Tim Dolbeare, Melise Edwards, John Galbraith, Nathalie Gaudreault, Terri L Gilbert, Fiona Griffin, Perry Hargrave, Robert Howard, Lawrence Huang, Sean Jewell, Nika Keller, Ulf Knoblich, Josh D Larkin, Rachael Larsen, Chris Lau, Eric Lee, Felix Lee, Arielle Leon, Lu Li, Fuhui Long, Jennifer Luviano, Kyla Mace, Thuyanh Nguyen, Jed Perkins, Miranda Robertson, Sam Seid, Eric Shea-Brown, Jianghong Shi, Nathan Sjoquist, Cliff Slaughterbeck, David Sullivan, Ryan Valenza, Casey White, Ali Williford, Daniela M Witten, Jun Zhuang,

Hongkui Zeng, Colin Farrell, Lydia Ng, Amy Bernard, John W Phillips, R Clay Reid, and Christof Koch. A large-scale standardized physiological survey reveals functional organization of the mouse visual cortex. *Nature neuroscience*, 23(1):138–151, 2020.

[8] P Elias. Predictive coding–i. *I.R.E. transactions on information theory*, 1(1):16–24, 1955.

[9] K Fukushima. Neocognitron: a self organizing neural network model for a mechanism of pattern recognition unaffected by shift in position. *Biological cybernetics*, 36(4):193–202, 1980.

[10] Rohan Gala, Agata Budzillo, Fahimeh Baftizadeh, Jeremy Miller, Nathan Gouwens, Anton Arkhipov, Gabe Murphy, Bosiljka Tasic, Hongkui Zeng, Michael Hawrylycz, and Uygar Sümbül. Consistent cross-modal identification of cortical neurons with coupled autoencoders. *bioRxiv*, 2021.

[11] Felix A Gers, Jürgen Schmidhuber, and Fred Cummins. Learning to forget: Continual prediction with lstm. *Neural computation*, 12(10):2451–2471, 2000.

[12] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep learning*. Adaptive computation and machine learning. The MIT Press, London, England, 2016.

[13] Nathan W. Gouwens, Staci A. Sorensen, Fahimeh Baftizadeh, Agata Budzillo, Brian R. Lee, Tim Jarsky, Lauren Alfiler, Anton Arkhipov, Katherine Baker, Eliza Barkan, Kyla Berry, Darren Bertagnolli, Kris Bickley, Jasmine Bomben, Thomas Braun, Krissy Brouner, Tamara Casper, Kirsten Crichton, Tanya L. Daigle, Rachel Dalley, Rebecca de Frates, Nick Dee, Tsega Desta, Samuel Dingman Lee, Nadezhda Dotson, Tom Egdorf, Lauren Ellingwood, Rachel Enstrom, Luke Esposito, Colin Farrell, David Feng, Olivia Fong, Rohan Gala, Clare Gamlin, Amanda Gary, Alexandra Glandon, Jeff Goldy, Melissa Gorham, Lucas Graybuck, Hong Gu, Kristen Hadley, Michael J. Hawrylycz, Alex M. Henry, DiJon Hill, Madie Hupp, Sara Kebede, Tae Kyung Kim, Lisa Kim, Matthew Kroll, Changkyu Lee, Katherine E. Link, Matthew Mallory, Rusty Mann, Michelle Maxwell, Medea McGraw, Delissa McMillen, Alice Mukora, Lindsay Ng, Lydia Ng, Kiet Ngo, Philip R. Nicovich, Aaron Oldre, Daniel Park, Hanchuan Peng, Osnat Penn, Thanh Pham, Alice Pom, Lydia Potekhina, Ramkumar Rajanbabu, Shea Ransford, David Reid, Christine Rimorin, Miranda Robertson, Kara Ronellenfitch, Augustin Ruiz, David Sandman, Kimberly Smith, Josef Sulc, Susan M. Sunkin, Aaron Szafer, Michael Tieu, Amy Torkelson, Jessica Trinh, Herman Tung, Wayne Wakeman, Katelyn Ward, Grace Williams, Zhi Zhou, Jonathan Ting, Uygar Sumbul, Ed Lein, Christof Koch, Zizhen Yao, Bosiljka Tasic, Jim Berg, Gabe J. Murphy, and Hongkui Zeng. Toward an integrated classification of neuronal cell types: morphoelectric and transcriptomic characterization of individual gabaergic cortical neurons. *bioRxiv*, 2020.

[14] Nathan W Gouwens, Staci A Sorensen, Jim Berg, Changkyu Lee, Tim Jarsky, Jonathan Ting, Susan M Sunkin, David Feng, Costas A Anastassiou, Eliza Barkan, Kris Bickley, Nicole Blesie, Thomas Braun, Krissy Brouner, Agata Budzillo, Shiella Caldejon, Tamara Casper, Dan Castelli, Peter Chong, Kirsten Crichton, Christine Cuhaciyan, Tanya L Daigle, Rachel Dalley, Nick Dee, Tsega Desta, Song-Lin Ding, Samuel Dingman, Alyse Doperalski, Nadezhda Dotson, Tom Egdorf, Michael Fisher, Rebecca A de Frates, Emma Garren, Marissa Garwood, Amanda Gary, Nathalie Gaudreault, Keith Godfrey, Melissa Gorham, Hong Gu, Caroline Habel, Kristen Hadley, James Harrington, Julie A Harris, Alex Henry, DiJon Hill, Sam Josephsen, Sara Kebede, Lisa Kim, Matthew Kroll, Brian Lee, Tracy Lemon, Katherine E Link, Xiaoxiao Liu, Brian Long, Rusty Mann, Medea McGraw, Stefan Mihalas, Alice Mukora, Gabe J Murphy, Lindsay Ng, Kiet Ngo, Thuc Nghi Nguyen, Philip R Nicovich, Aaron Oldre, Daniel Park, Sheana Parry, Jed Perkins, Lydia Potekhina, David Reid, Miranda Robertson, David Sandman, Martin Schroedter, Cliff Slaughterbeck, Gilberto Soler-Llavina, Josef Sulc, Aaron Szafer, Bosiljka Tasic, Naz Taskin, Corinne Teeter, Nivretta Thatra, Herman Tung, Wayne Wakeman, Grace Williams, Rob Young, Zhi Zhou, Colin Farrell, Hanchuan Peng, Michael J Hawrylycz, Ed Lein, Lydia Ng, Anton Arkhipov, Amy Bernard, John W Phillips, Hongkui Zeng, and Christof Koch. Classification of electrophysiological and morphological neuron types in the mouse visual cortex. *Nature neuroscience*, 22(7):1182–1195, 2019.

[15] Julie A Harris, Stefan Mihalas, Karla E Hirokawa, Jennifer D Whitesell, Hannah Choi, Amy Bernard, Phillip Bohn, Shiella Caldejon, Linzy Casal, Andrew Cho, Aaron Feiner, David Feng, Nathalie Gaudreault, Charles R Gerfen, Nile Graddis, Peter A Groblewski, Alex M Henry, Anh Ho, Robert Howard, Joseph E Knox, Leonard Kuan, Xiuli Kuang, Jerome Lecoq, Phil Lesnar, Yaoyao Li, Jennifer Luviano, Stephen McConoughey, Marty T Mortrud, Maitham Naeemi, Lydia Ng, Seung Wook Oh, Benjamin Ouellette, Elise Shen, Staci A Sorensen, Wayne Wakeman, Quanxin Wang, Yun Wang, Ali Williford, John W Phillips, Allan R Jones, Christof Koch, and Hongkui Zeng. Hierarchical organization of cortical and thalamic connectivity. *Nature (London)*, 575(7781):195–202, 2019.

[16] Matthias H Hennig. Theoretical models of synaptic short term plasticity. *Frontiers in computational neuroscience*, 7:45–45, 2013.

[17] Sepp Hochreiter and Jürgen Schmidhuber. Long short-term memory. *Neural computation*, 9(8):1735–1780, 1997.

[18] Brian Hu, Marina E. Garrett, Peter A. Groblewski, Douglas R. Ollerenshaw, Jiaqi Shang, Kate Roll, Sahar Manavi, Christof Koch, Shawn R. Olsen, and Stefan Mihalas. Adaptation supports short-term memory in a visual change detection task. *bioRxiv*, 2020.

[19] Yanping Huang and Rajesh P. N Rao. Predictive coding. *Wiley interdisciplinary reviews. Cognitive science*, 2(5):580–593, 2011.

[20] D. H Hubel and T. N Wiesel. Receptive fields of single neurones in the cat's striate cortex. *The Journal of physiology*, 148(3):574–591, 1959.

[21] Peter J. Huber. *Robust statistics*. Wiley, Hoboken, N.J., second edition. edition, 2009.

[22] D.J Krusienski and K Jenkins. Comparative analysis of neural network filters and adaptive volterra filters. In *Proceedings of the 44th IEEE 2001 Midwest Symposium on Circuits and Systems. MWSCAS 2001 (Cat. No.01CH37257)*, volume 1, pages 49–52 vol.1. IEEE, 2001.

[23] Jose Nathan Kutz. *Data-driven modeling & scientific computation: methods for complex systems & big data*. Oxford University Press, 2013.

[24] Yann LeCun and Corinna Cortes. MNIST handwritten digit database. 2010.

[25] Jung Hoon Lee, Luke Campagnola, Stephanie C. Seeman, Tim Jarsky, and Stefan Mihalas. Functional synapse types via characterization of short-term synaptic plasticity. *bioRxiv*, 2019.

[26] Sandrine Lefort and Carl C H Petersen. Layer-dependent short-term synaptic plasticity between excitatory neurons in the c2 barrel column of mouse primary somatosensory cortex. *Cerebral cortex (New York, N.Y. 1991)*, 27(7):3869–3878, 2017.

[27] William Lotter, Gabriel Kreiman, and David D. Cox. Deep predictive coding networks for video prediction and unsupervised learning. *CoRR*, abs/1605.08104, 2016.

[28] Nicolas Y Masse, Guangyu R Yang, H Francis Song, Xiao-Jing Wang, and David J Freedman. Circuit mechanisms for the maintenance and manipulation of information in working memory. *Nature neuroscience*, 22(7):1159–1167, 2019.

[29] Nicolas Perez-Nieves, Vincent C. H. Leung, Pier Luigi Dragotti, and Dan F. M. Goodman. Neural heterogeneity promotes robust learning. *bioRxiv*, 2021.

[30] Rajesh P. N Rao and Dana H Ballard. Predictive coding in the visual cortex: a functional interpretation of some extra-classical receptive-field effects. *Nature neuroscience*, 2(1):79–87, 1999.

[31] Alex Reyes and Bert Sakmann. Developmental switch in the short-term modification of unitary epsps evoked in layer 2/3 and layer 5 pyramidal neurons of rat neocortex. *The Journal of neuroscience*, 19(10):3827–3835, 1999.

[32] Magnus J. E Richardson, Ofer Melamed, Gilad Silberberg, Wulfram Gerstner, and Henry Markram. Short-term synaptic plasticity orchestrates the response of pyramidal cells and interneurons to population bursts. *Journal of computational neuroscience*, 18(3):323–331, 2005.

[33] Stephanie C Seeman, Luke Campagnola, Pasha A Davoudian, Alex Hoggarth, Travis A Hage, Alice Bosma-Moody, Christopher A Baker, Jung Hoon Lee, Stefan Mihalas, Corinne Teeter, Andrew L Ko, Jeffrey G Ojemann, Ryder P Gwinn, Daniel L Silbergeld, Charles Cobbs, John Phillips, Ed Lein, Gabe Murphy, Christof Koch, Hongkui Zeng, and Tim Jarsky. Sparse recurrent excitatory connectivity in the microcircuit of the adult mouse and human cortex. *eLife*, 7, 2018.

[34] David Sussillo and Omri Barak. Opening the black box: low-dimensional dynamics in high-dimensional recurrent neural networks. *Neural computation*, 25(3):626–649, 2013.

[35] Bosiljka Tasic, Vilas Menon, Thuc Nghi Nguyen, Tae Kyung Kim, Tim Jarsky, Zizhen Yao, Boaz Levi, Lucas T Gray, Staci A Sorensen, Tim Dolbeare, Darren Bertagnolli, Jeff Goldy, Nadiya Shapovalova, Sheana Parry, Changkyu Lee, Kimberly Smith, Amy Bernard, Linda Madisen, Susan M Sunkin, Michael Hawrylycz, Christof Koch, and Hongkui Zeng. Adult mouse cortical cell taxonomy revealed by single cell transcriptomics. *Nature neuroscience*, 19(2):335–346, 2016.

[36] Bosiljka Tasic, Zizhen Yao, Lucas T Graybuck, Kimberly A Smith, Thuc Nghi Nguyen, Darren Bertagnolli, Jeff Goldy, Emma Garren, Michael N Economo, Sarada Viswanathan, Osnat Penn, Trygve Bakken, Vilas Menon, Jeremy Miller, Olivia Fong, Karla E Hirokawa, Kanan Lathia, Christine Rimorin, Michael Tieu, Rachael Larsen, Tamara Casper, Eliza Barkan, Matthew Kroll, Sheana Parry, Nadiya V Shapovalova, Daniel Hirschstein, Julie Pendergraft, Heather A Sullivan, Tae Kyung Kim, Aaron Szafer, Nick Dee, Peter Groblewski, Ian Wickersham, Ali Cetin, Julie A Harris, Boaz P Levi, Susan M Sunkin, Linda Madisen, Tanya L Daigle, Loren Looger, Amy Bernard, John Phillips, Ed Lein, Michael Hawrylycz, Karel Svoboda, Allan R Jones, Christof Koch, and Hongkui Zeng. Shared and distinct transcriptomic cell types across neocortical areas. *Nature (London)*, 563(7729):72–78, 2018.

[37] Corinne Teeter, Ramakrishnan Iyer, Vilas Menon, Nathan Gouwens, David Feng, Jim Berg, Aaron Szafer, Nicholas Cain, Hongkui Zeng, Michael Hawrylycz, Christof Koch, and Stefan Mihalas. Generalized leaky integrate-and-fire models classify multiple neuron types. *Nature communications*, 9(1):709–709, 2018.

[38] Stephen Wiggins. *Introduction to applied nonlinear dynamical systems and chaos*. Texts in applied mathematics ; 2. Springer, New York, second edition. edition, 2003.

[39] Yong Yu, Xiaosheng Si, Changhua Hu, and Jianxun Zhang. A review of recurrent neural networks: Lstm cells and network architectures. *Neural computation*, 31(7):1235–1270, 2019.

[40] Anthony Zaknich. *Principles of adaptive filters and self-learning systems*. Advanced textbooks in control and signal processing. Springer London, London], 2005.

[41] Aston Zhang, Zachary C. Lipton, Mu Li, and Alexander J. Smola. *Dive into Deep Learning*. 2020. https://d2l.ai.

[42] Chengxu Zhuang, Siming Yan, Aran Nayebi, Martin Schrimpf, Michael C Frank, James J DiCarlo, and Daniel L K Yamins. Unsupervised neural network models of the ventral visual stream. *Proceedings of the National Academy of Sciences - PNAS*, 118(3), 2021.

[43] Robert S Zucker and Wade G Regehr. Short-term synaptic plasticity. *Annual review of physiology*, 64(1):355–405, 2002.

Appendix A

SUPPLEMENTARY MATERIAL

A.0.1 Model Trainable Parameters

Tables A.1-A.3 detail the hidden layer dimensions and trainable parameter counts for the implemented models. Table A.4 details the trainable parameter counts for simulations described in Figure 2.9, for which the number of hidden units was conserved across model type. Table A.5 details the trainable parameter counts for simulations described in Figure 2.11, for which the number of hidden units was set at 256.

Table A.1: The parameter matched hidden layer sizes and corresponding trainable parameter counts for an input layer dimension of 4. Relevant for models presented in Figure 2.3A.

Model	**Hidden Layer Size**	**Trainable Parameters**
RNN	16	420
LSTM	7	396
STSP	32	301
RNSTSP	16	420
RNSTSPtau	16	436
STSPtau	32	305
STSPdiv	32	301

Table A.2: The parameter matched hidden layer sizes and corresponding trainable parameter counts for an input layer dimension of 8. Relevant for models presented in Figure 2.3B.

Model	**Hidden Layer Size**	**Trainable Parameters**
RNN	16	552
LSTM	7	540
STSP	32	552
RNSTSP	16	552
RNSTSPtau	16	568
STSPtau	32	569
STSPdiv	32	552

Table A.3: The parameter matched hidden layer sizes and corresponding trainable parameter counts for an input layer dimension of 16. Relevant for models presented in Figure 2.3C.

Model	**Hidden Layer Size**	**Trainable Parameters**
RNN	16	816
LSTM	7	828
STSP	24	808
RNSTSP	16	816
RNSTSPtau	16	832
STSPtau	24	824
STSPdiv	24	824

Table A.4: Trainable parameter counts for models with matched hidden layers dimensions of 16 nodes.

Model	**Input Dimension**	**Trainable Parameters**
RNN	4	420
LSTM	4	1476
STSP	4	148
RNSTSP	4	420
RNSTSPtau	4	436
STSPtau	4	152
STSPdiv	4	148
RNN	8	552
LSTM	8	1800
STSP	8	280
RNSTSP	8	552
RNSTSPtau	8	568
STSPtau	8	288
STSPdiv	8	280
RNN	16	816
LSTM	16	2448
STSP	16	544
RNSTSP	16	816
RNSTSPtau	16	832
STSPtau	16	560
STSPdiv	16	544

Table A.5: Trainable parameter counts for models with 256 node hidden layers. Relevant for models presented in Figure 2.11

Model	**Input Dimension**	**Trainable Parameters**
RNN	4	68100
LSTM	4	269316
STSP	4	2308
RNSTSP	4	68100
RNSTSPtau	4	68356
STSPtau	4	2312
STSPdiv	4	2308
RNN	8	70152
LSTM	8	274440
STSP	8	4360
RNSTSP	8	70152
RNSTSPtau	8	70408
STSPtau	8	4368
STSPdiv	8	4360
RNN	16	74256
LSTM	16	284688
STSP	16	8464
RNSTSP	16	74256
RNSTSPtau	16	74512
STSPtau	16	8480
STSPdiv	16	8464

Appendix B

PYTHON CODE SAMPLES

```
class STSPnet(nn.Module):
    def __init__(self, tau_init,
                 tau_x=3, # x recovery time constant
                 syn_u=0.5, # calcium concentration
                 input_size=2,
                 hidden_dim=10,
                 output_size=2,
                 noise_std = 0.1,
                 STSP_perc = 1.0,
                 device = 'cpu'):
        super(STSPnet, self).__init__()
        self.device = device

        self.syn_u = syn_u
        self.input_size = input_size

        #initializing synaptic dynamics time constants
        if tau_init == 'rand_uniform':
            #randomly initialize the time constants for each hidden neuron
            epsilon = np.finfo(np.float).eps#get machine epsilon for low end of
                uniform dist
            tau_x_np = np.random.uniform(epsilon, tau_x, (1,
                input_size)).astype(np.float32)
            self.tau_x = torch.from_numpy(tau_x_np).to(self.device)
```

```
23
24          elif tau_init == 'rand_normal':
25              mu1, sigma1 = 2.5, 1.2#set normal dist characteristics
26              tau_x_np = np.random.uniform(mu1, sigma1, (1,
                    input_size)).astype(np.float32)
27              self.tau_x = torch.from_numpy(tau_x_np).to(self.device)
28
29          elif tau_init == 'fixed_dist':
30              if torch.is_tensor(tau_x):
31                  if tau_x.shape[0] == input_size:
32                      self.tau_x = tau_x.to(self.device)
33                  else:
34                      raise ValueError('tau_x tensor is not of size input_size')
35              else:
36                  raise ValueError('tau_x is not a torch tensor of size input_size')
37
38          else:
39             self.tau_x = tau_x
40
41          self.STSP_perc = STSP_perc#percentage of STSP neurons in RNN
42          #create mask to identify STSP and non-STSP neurons
43          total_neurons = self.input_size
44          STSP_neuron_count = int(total_neurons*self.STSP_perc)#number of nodes
                with STSP connections
45          self.nonSTSP_neuron_count = total_neurons - STSP_neuron_count#number of
                nodes with STSP connections
46          #randomly assign neurons as non-STSP across the layer up to the
                appropriate total count
```

```
47          flat_indices = np.random.choice(np.arange(self.input_size),
                replace=False, size=self.nonSTSP_neuron_count)#get flattened array
                indexes
48          self.nonSTSP_indices = np.unravel_index(flat_indices, (1,
                self.input_size))#get tuple of array containing index along each axis
49
50
51          if noise_std is not None:
52              self.noise = tdist.Normal(torch.tensor([0.0]),
                    torch.tensor([noise_std]))
53          else:
54              self.noise = None
55
56          self.linear_1 = nn.Linear(input_size, hidden_dim)
57          self.linear_2 = nn.Linear(hidden_dim, output_size)
58
59      def forward(self, inputs, syn_x):
60
61          ### Update synaptic plasticity ###
62          syn_x = syn_x + (1-syn_x)/self.tau_x - self.syn_u * syn_x * inputs
63          syn_x = torch.clamp(syn_x, min=0, max=1) # make sure between [0,1]
64          # import matplotlib.pyplot as plt
65          # plt.plot(torch.reshape(inputs, (50, 4)))
66          # plt.show()
67          #if the connection is masked as non-stsp... set synaptic resources
                variable to 1
68          batch_size = inputs.shape[1]
69          nonSTSP_indices_batched = (np.repeat(self.nonSTSP_indices[0],batch_size),
                np.repeat(np.arange(batch_size), self.nonSTSP_neuron_count),
```

```
            np.repeat(self.nonSTSP_indices[1],batch_size))
        syn_x[nonSTSP_indices_batched] = 1.0#set synaptic resources to 1.0 for
            non-STSP neurons so they behave like standard RNN nodes

        if self.noise is not None:
            noise = self.noise.sample((self.input_size,)).view
                (1,self.input_size).to(self.device)
            h = syn_x * (inputs+noise) # effective inputs after STSPnet
        else:
            h = syn_x * inputs


        output = F.relu(self.linear_1(h))
        h_ = output
        output = self.linear_2(output)
        #output = F.relu(self.linear_2(output))

        # output = torch.tanh(self.linear_1(h))
        # output = torch.tanh(self.linear_2(output))

        return output, syn_x, h_

    def init_hidden(self):
        """Initialize syn_x for the inputs units."""
        #return torch.randn([1,
            self.input_size],dtype=torch.float).to(self.device)
        return torch.ones([1, self.input_size],dtype=torch.float).to(self.device)
```

```
class RNSTSP_tau(nn.Module):

```

```
    def __init__(self, tau_init,
                 tau_max=4, # x recovery time constant
                 syn_u=0.5, # calcium concentration
                 input_size=2,
                 hidden_dim=10,
                 output_size=2,
                 n_layers=1,
                 noise_std = 0.1,
                 STSP_perc = 1.0,
                 init_style = None,
                 device = 'cpu'):

        super(RNSTSP_tau, self).__init__()
        self.device = device
        # Defining some parameters
        self.tau_max = tau_max
        self.syn_u = syn_u
        self.input_size = input_size
        self.hidden_dim = hidden_dim
        self.n_layers = n_layers
        self.STSP_perc = STSP_perc#percentage of STSP neurons in RNN

        #create mask to identify STSP and non-STSP neurons
        total_neurons = self.n_layers*self.hidden_dim
        STSP_neuron_count = int(total_neurons*self.STSP_perc)#number of nodes
            with STSP connections
        self.nonSTSP_neuron_count = total_neurons - STSP_neuron_count#number of
            nodes with STSP connections
        #randomly assign neurons as non-STSP across the layer up to the
```

```
            appropriate total count
30          flat_indices = np.random.choice(np.arange(self.n_layers*self.hidden_dim),
                replace=False, size=self.nonSTSP_neuron_count)#get flattened array
                indexes
31          self.nonSTSP_indices = np.unravel_index(flat_indices, (self.n_layers,
                self.hidden_dim))#get tuple of array containing index along each axis
32
33          if noise_std is not None:
34              self.noise = tdist.Normal(torch.tensor([0.0]),
                    torch.tensor([noise_std]))
35          else:
36              self.noise = None
37
38          if tau_init == 'rand_uniform':
39              #randomly initialize the time constants for each hidden neuron
40              epsilon = np.finfo(np.float).eps#get machine epsilon for low end of
                    uniform dist
41              tau_x_np = np.random.uniform(epsilon, tau_max, (1,
                    hidden_dim)).astype(np.float32)
42              self.tau_x = nn.Parameter(torch.from_numpy(tau_x_np),
                    requires_grad=True)
43          elif tau_init == 'rand_bimodal':
44              bimodal_id = np.random.randint(2, size=hidden_dim)#get a random
                    bimodal id for each hidden node
45              mu1, sigma1 = 1, 0.25#set bimodal dist 1 characteristics
46              mu2, sigma2 = 4, 0.25#set bimodal dist 2 characteristics
47              init_vals_list = []
48              for i in range(len(bimodal_id)):
49                  if bimodal_id[i] == 0:
```

```
50                  init_val = np.random.normal(mu1, sigma1)
51              else:
52                  init_val = np.random.normal(mu2, sigma2)
53              init_vals_list.append(init_val)
54
55          init_vals =
                np.array(init_vals_list).astype(np.float32).reshape(1,hidden_dim)
56          self.tau_x = nn.Parameter(torch.from_numpy(init_vals),
                requires_grad=True)#.to(self.device)
57
58      else:
59          self.tau_x = nn.Parameter(torch.ones(1, hidden_dim),
                requires_grad=True)#.to(self.device)
60
61
62      #Defining the layers
63      self.rnn = nn.RNN(input_size, hidden_dim, n_layers, nonlinearity='relu',
            batch_first = True)# RNN Layer
64
65      self.linear_2 = nn.Linear(hidden_dim, output_size)
66      self.init_rnn_weights(style=init_style)
67
68  def forward(self, inputs, hidden, syn_x):
69
70      ### Update synaptic plasticity ###
71      #self.tau_x = nn.Parameter(torch.clamp(self.tau_x,
            min=torch.finfo(torch.float32).eps))#make sure tau_x is greater than
            zero
72      syn_x = syn_x + (1-syn_x)/self.tau_x - self.syn_u * syn_x * hidden
```

```
73          syn_x = torch.clamp(syn_x, min=0, max=1) # make sure between [0,1]
74
75          batch_size = hidden.shape[1]
76          nonSTSP_indices_batched = (np.repeat(self.nonSTSP_indices[0],batch_size),
                np.repeat(np.arange(batch_size), self.nonSTSP_neuron_count),
                np.repeat(self.nonSTSP_indices[1],batch_size))
77          syn_x[nonSTSP_indices_batched] = 1.0#set synaptic resources to 1.0 for
                non-STSP neurons so they behave like standard RNN nodes
78
79          if self.noise is not None:
80              noise = self.noise.sample((self.input_size,)).view
                    (1,self.input_size).to(self.device)
81              h = syn_x * (hidden + noise) # effective inputs after STSPnet
82          else:
83              h = syn_x * hidden
84
85          self.rnn.flatten_parameters()
86          output, hidden = self.rnn(inputs, h)
87          h_ = output
88          #passing RNN output through to output layer
89          output = self.linear_2(output)
90          #output = F.relu(self.linear_2(output))
91          #output = torch.tanh(self.linear_2(output))
92
93          return output, hidden, syn_x, h_
94
95      def init_rnn_weights(self, style):
96
97          if style == 'Vartak':
```

```
98          N = self.hidden_dim
99
100         for name, param in self.rnn.named_parameters():
101             if 'weight_ih' in name:
102                 #input weights
103                 #Gaussian distribution with zero mean and variance 1/N.
                        Motivated by findings from
104                 #(Sussillo & Abbott, 2015), also scale Whx by a factor of a,
105                 a = math.sqrt(2)*math.exp(1.2/(max(N, 6) - 2.4))
106                 std = np.sqrt(1/N)
107                 nn.init.normal_(param.data, mean = 0.0, std = std)
108                 param.data = param.data*a
109
110             elif 'weight_hh' in name:
111                 #recurrent
112                 R = torch.normal(mean = 0.0, std=1.0, size = (N, N))
113                 R_T = torch.transpose(R, 0, 1)
114                 A = (1/N)*torch.matmul(R, R_T)
115                 eigs, evs = torch.eig(A, eigenvectors=False)
116                 eig_max = torch.max(eigs)
117                 Whh = A / eig_max
118                 param.data = Whh
119
120             elif 'bias' in name:
121                 pass
122
123
124         for name, param in self.linear_2.named_parameters():
125             if 'weight' in name:
```

```
126                 #output weights
127                 fan_in, fan_out =
                        nn.init._calculate_fan_in_and_fan_out(param.data)
128                 out_std = np.sqrt(2 / (fan_in + fan_out))
129                 nn.init.normal_(param.data, mean = 0.0, std = out_std)
130         else:
131             pass
132
133     def init_hidden(self, batch_size):
134         """Initialize hiden unit activations for the reccurent net."""
135         return torch.randn(self.n_layers, batch_size,
                self.hidden_dim).to(self.device)
136
137     def init_synaptic_resources(self):
138         """Initialize syn_x for the reccurent net units."""
139         #return torch.randn([1,
                self.hidden_dim],dtype=torch.float).to(self.device)
140         return torch.ones([1, self.hidden_dim],dtype=torch.float).to(self.device)
```