Deep Multi-State Dynamic Recurrent Neural Networks Operating on Wavelet Based Neural Features for Robust Brain Machine Interfaces

Benyamin Haghi\textsuperscript{1,*}, Spencer Kellis\textsuperscript{2}, Sahil Shah\textsuperscript{1}, Maitreyi Ashok\textsuperscript{1}, Luke Bashford\textsuperscript{2}, Daniel Kramer\textsuperscript{3}, Brian Lee\textsuperscript{3}, Charles Liu\textsuperscript{3}, Richard A. Andersen\textsuperscript{2}, Azita Emami\textsuperscript{1}

\textsuperscript{1} Electrical Engineering Department, Caltech, Pasadena, CA, USA
\textsuperscript{2} Biology and Biological Engineering Department, Caltech, Pasadena, CA, USA
\textsuperscript{3} Neurorestoration Center and Neurosurgery, USC Keck School of Medicine, L.A., CA, USA

*benyamin.a.haghi@caltech.edu

Abstract

We present a new deep multi-state Dynamic Recurrent Neural Network (DRNN) architecture for Brain Machine Interface (BMI) applications. Our DRNN is used to predict Cartesian representation of a computer cursor movement kinematics from open-loop neural data recorded from the posterior parietal cortex (PPC) of a human subject in a BMI system. We design the algorithm to achieve a reasonable trade-off between performance and robustness, and we constrain memory usage in favor of future hardware implementation. We feed the predictions of the network back to the input to improve prediction performance and robustness. We apply a scheduled sampling approach to the model in order to solve a statistical distribution mismatch between the ground truth and predictions. Additionally, we configure a small DRNN to operate with a short history of input, reducing the required buffering of input data and number of memory accesses. This configuration lowers the expected power consumption in a neural network accelerator. Operating on wavelet-based neural features, we show that the average performance of DRNN surpasses other state-of-the-art methods in the literature on both single- and multi-day data recorded over 43 days. Results show that multi-state DRNN has the potential to model the nonlinear relationships between the neural data and kinematics for robust BMIs.

1 Introduction

Brain-machine interfaces (BMIs) can help spinal cord injury (SCI) patients by decoding neural activity into useful control signals for guiding robotic limbs, computer cursors, or other assistive devices \[1\]. BMI in its most basic form maps neural signals into movement control signals and then closes the loop to enable direct neural control of movements. Such systems have shown promise in helping SCI patients. However, improving performance and robustness of these systems remains challenging. Even for simple movements, such as moving a computer cursor to a target on a computer screen, decoding performance can be highly variable over time. Furthermore, most BMI systems currently run on high-power computer systems. Clinical translation of these systems will require decoders that can adapt to changing neural conditions and which operate efficiently enough to run on mobile, even implantable, platforms.

Conventionally, linear decoders have been used to find the relationship between kinematics and neural signals of the motor cortex. For instance, Wu et al. \[2\] use a linear model to decode the neural activity of two macaque monkeys. Orsborn et al. \[3\] apply a Kalman filter, updating the model on batches of neural data of an adult monkey, to predict kinematics in a center-out task. Gilja et al. \[4\] propose a Kalman Filter to predict hand movement velocities of a monkey in a center-out task. However, all of these algorithms can only predict piecewise linear relationships between the neural data and kinematics. Moreover, because of nonstationarity and low signal-to-noise ratio (SNR) in the neural data, linear decoders need to be regularly re-calibrated \[2\].
Recently, nonlinear machine learning algorithms have shown promise in attaining high performance and robustness in BMIs. For instance, Wessberg et al. [5] apply a fully-connected neural network to neural data recorded from a monkey. Shpigelman et al. [6] show that a Gaussian kernel outperforms a linear kernel in a Kernel Auto-Regressive Moving Average (KARMA) algorithm when decoding 3D kinematics from macaque neural activity. Sussillo et al. [7] apply a large FORCE Dynamic Recurrent Neural Network (F-DRNN) on neural data recorded from the primary motor cortex in two monkeys, and then they test the stability of the model over multiple days [8]. Zhang et al. [9] and Schwemmer et al. [10] extract wavelet based features of motor cortex neural data of a human subject to classify intended hand movements by using a nonlinear support vector machine (SVM) and a large deep neural network, respectively. Hosman et al. [11] pass motor cortex neural firing rates to an LSTM and a Kalman filter to compare their performances for decoding intended cursor velocity of a human subject. These nonlinear learning-based decoders have shown more stability over multiple days and have improved performance compared to prior linear methods. However, they all have been applied to motor cortex data by mostly using neural firing rates as input features, which show more variability over long periods [2]. Recent work has demonstrated that neural activity in the posterior parietal cortex (PPC) can be used to support BMIs [12, 13, 14, 15, 16, 17, 18], although the encoding of movement kinematics appears to be complex. PPC processes a rich set of high-level aspects of movement including sensory integration, planning, and execution [13] and may encode this information differently [15]. These characteristics of PPC differentiate it from other brain areas and, while providing a large amount of information to the decoder, also require new paradigms, such as those discussed here, to extract useful information. Therefore, extracting appropriate neural features and designing a robust decoder that can model this relationship in an actual BMI setting is required.

We propose a new Deep Multi-State Dynamic Recurrent Neural Network (DRNN) decoder to address the challenges of performance, robustness, and potential hardware implementation. We refer to two theorems to show the stability, convergence, and potential of DRNNs for approximation of state-space trajectories (see supplementary material). We train the DRNN by passing a history of input data to it and feeding the predictions of the system back to the input to improve performance and robustness for sequential data prediction. Moreover, we apply scheduled sampling to solve the statistical distribution discrepancy between the ground truth and predictions. By extracting different neural features, we compare the performance and robustness of the DRNN with the existing methods in the literature to predict hand movement kinematics from open-loop neural data. Our BMI data are recorded from the PPC of a human subject over 43 days. Finally, we discuss the potential for implementing our DRNN efficiently in hardware for implantable platforms. To the best of our knowledge, this is the first demonstration of applying learning-based decoders to a human PPC activity. Our results indicate that the Deep Multi-State DRNN operating on mid-band wavelet-based neural features has the potential to model the nonlinear relationships between the neural data and kinematics for robust BMIs.

2 Deep multi-state dynamic recurrent neural network

A DRNN is a nonlinear dynamic system described by a set of differential or difference equations. It contains both feed-forward and feedback synaptic connections. In addition to the recurrent architecture, a nonlinear and dynamic structure enables it to capture time-varying spatiotemporal relationships in the sequential data. Moreover, because of state feedback, a small recurrent network can be equivalent to a large feed-forward network. Therefore, a recurrent network will be computationally efficient, especially for the applications that require hardware implementation [19]. We define our deep multi-state DRNN at each time step \( k \) as below:

\[
\begin{align*}
  s_k &= W_{ss}s_{k-1} + W_{sr}r_{k-1} + W_{su}u_k + W_{sf}z_k + b_s \\
  r_k &= \tanh(s_k) \\
  h_k^{(1)} &= \tanh(W_{h(1)s}h_k^{(1)} + W_{h(1)r}r_k + b_{h(1)}) \\
  h_k^{(i)} &= \tanh(W_{h(i)s}h_k^{(i-1)} + W_{h(i)r}r_k^{(i-1)} + b_{h(i)}) \\
  \hat{y}_k &= W_{yh(i)}h_k^{(i)} + b_y \\
  \hat{y}_k &= \tanh(\hat{y}_k), \ |\hat{y}_k| > 1 \\
  z_k &\leftarrow \hat{y}_k \text{ or } y_k \text{ (Scheduled Sampling)}
\end{align*}
\]

\( s \in \mathbb{R}^N \) is the activation variable, and \( r \in \mathbb{R}^N \) is the vector of corresponding firing rates. These two internal states track the first- and zero-order differential features of the system, respectively. Unlike
We evaluate the performance of our DRNN on 12 neural features: High-frequency, Mid-frequency, LFADS features \[23\]; High-Pass and Low-Pass Filtered (HPF, LPF) data; Threshold Crossings To generate TCs, we threshold bandpass-filtered (250Hz - 5KHz) neural data at -4 times the root-mean

3 Pre-processing and feature engineering

We evaluate the performance of our DRNN on 12 neural features: High-frequency, Mid-frequency, and Low-frequency Wavelet features (HWT, MWT, LWT); High-frequency, Mid-frequency, and Low-frequency Fourier powers (HFT, MFT, LFT); Latent Factor Analysis via Dynamical Systems (LFADS) features \[23\]; High-Pass and Low-Pass Filtered (HPF, LPF) data; Threshold Crossings (TCs); Multi-Unit Activity (MUA); and combined MWT and TCs (MWT + TCs) (Table 1).

To extract wavelet features, we use 'db4' mother wavelet on 50ms moving windows of the voltage time series recorded from each channel. Then, the mean of absolute-valued coefficients for each scale is calculated to generate 11 time series for each channel. HWT is formed from the wavelet scales 1 and 2 (effective frequency range \(\geq 3.75KHz\)). MWT is made from the wavelet scales 3 to 6 (234Hz - 3.75KHz). Finally, LWT shows the activity of scales 7 to 11 as the low frequency scales (\(\leq 234\)Hz).

Fourier-based features are extracted by computing the Fourier transform with the sampling frequency of 30KHz on one-second moving windows for each channel. Then, the band-powers at the same 11 scales of the wavelet features are divided by the total power at the frequency band of 0Hz - 15KHz.

To generate TCs, we threshold bandpass-filtered (250Hz - 5KHz) neural data at -4 times the root-mean

\[1\] Our code is available at: https://github.com/BenyaminHaghi/DRNN-NeurIPS2019

3.75KHz). Finally, LWT shows the activity of scales 7 to 11 as the low frequency scales (\(\geq 234\)Hz).

Fourier-based features are extracted by computing the Fourier transform with the sampling frequency of 30KHz on one-second moving windows for each channel. Then, the band-powers at the same 11 scales of the wavelet features are divided by the total power at the frequency band of 0Hz - 15KHz.

To generate TCs, we threshold bandpass-filtered (250Hz - 5KHz) neural data at -4 times the root-mean
Algorithm 1 Training – DRNN with Feedback

1: Require: \( E, p_f, p_s \)
2: for \( e = 1 \) to \( E \) do
3: \( p_i = p_f - p_s E e + p_s \)
4: for \( i = 1 \) to number of batches do
5: Require: \( u, y \): Input and ground truth
6: if \( i = 1 \) then
7: \( z = y \)
8: end if
9: \( s \leftarrow N(0,\sigma_s) \), \( r \leftarrow \tanh(s) \)
10: if layers = 2 then
11: \( h = 0 \)
12: end if
13: for \( k = 2 \) to batch length do
14: \( s_k = W_{ss} s_{k-1} + W_{sr} r_{k-1} + W_{su} u_k + W_{sf} z_{k-1} + b_s \)
15: \( r_k = \tanh(s_k) \)
16: if layers = 1 then
17: \( \hat{y}_k = W_{yr} r_k + b_y \)
18: end if
19: else if layers = 2 then
20: \( h_k = \tanh(W_{hh} h_{k-1} + W_{hr} r_k + b_h) \)
21: \( \hat{y}_k = W_{yh} h_k + b_y \)
22: end if
23: if \( |\hat{y}_k| > 1 \) then
24: \( \hat{y}_k = \tanh(\hat{y}_k) \)
25: end if
26: \( z_k \leftarrow \hat{y}_k \) or \( y_k \) (Scheduled Sampling)
27: end for
28: Update weights and biases: BPTT
29: end for
30: end for
31: Until validation loss increases

---

**Table 1: Frequency range of features**

<table>
<thead>
<tr>
<th>Features</th>
<th>Frequency Range</th>
</tr>
</thead>
<tbody>
<tr>
<td>HWT, HFT, HPF</td>
<td>&gt; 3.75KHz</td>
</tr>
<tr>
<td>TCs, LFADS</td>
<td>250Hz - 5KHz</td>
</tr>
<tr>
<td>MWT, MFT, BPF</td>
<td>234Hz - 3.75KHz</td>
</tr>
<tr>
<td>LWT, LFT, LPF</td>
<td>&lt; 234Hz</td>
</tr>
</tbody>
</table>

---

4 Experimental Results

We conduct our FDA- and IRB-approved study of a BMI with a 32 year-old tetraplegic (C5-C6)
Figure 3: Architecture of our BMI system. Recorded neural activities of Anterior Intraparietal Sulcus (AIP), and Broadman’s Area 5 (BA5) are passed to a feature extractor. After pre-processing and feature selection, the data is passed to the decoder to predict the kinematics in a center-out task.

human research participant. This participant has Utah electrode arrays (NeuroPort, Blackrock Microsystems, Salt Lake City, UT, USA) implanted in the medial bank of Anterior Intraparietal Sulcus (AIP), and Broadman’s Area 5 (BA5). In a center-out task, a cursor moves, in two dimensions on a computer screen, from the center of a computer screen outward to one of eight target points located around a unit circle. A trial is one trajectory of the cursor from the center of the screen to one of the eight targets on a unit circle (Figure 5). During open-loop training, the participant observes the cursor move under computer control for 3 minutes. We collected open-loop training data from 66 blocks over 43 days for offline analysis of the DRNN. Broadband data were sampled at 30,000 samples/sec from the two implanted electrode arrays (96 channels each). Of the 43 total days, 42 contain 1 to 2 blocks of training data and 1 day contains 6 blocks, with about 50 trials per block. Moreover, these 43 days include 32, 5, 1, and 5 days of 2015, 2016, 2017, and 2018, respectively.

Since the predictions and the ground truth should be close in both micro and macro scales, we report root mean square error (RMSE) and $R^2$ as measures of average point-wise error and the strength of the linear association between the predicted and the ground truth signals, respectively. Results reported in the body of the paper are $R^2$ values for Y-axis position. $R^2$ values for X-axis position and velocities in X and Y directions and RMSE values for all the kinematics are all presented in supplementary material. All the curves and bar plots are shown by using 95% confidence intervals and standard deviations, respectively.

The available data is split into train and validation sets for parameter tuning. Parameters are computed on the training data and applied to the validation data. We perform 10 fold cross-validation by splitting the training data to 10 sets. Every time, the decoder is trained on 9 sets for different set of parameters and validated on the last set. We find the set of optimum parameters by using random search, as it has shown better performance than grid search [28]. Finally, we test the decoder with optimized parameters on the test set. The performance on all the test sets is averaged to report the overall performance of the models in both single- and multi-day analysis.

We compare our DRNN with other decoders, ranging from linear and historical decoders to nonlinear and modern techniques. The linear and historical decoders with which we compare ours are the Linear Model (LM) [2] and Kalman Filter (KF) [3]. The nonlinear and modern techniques with which we also compare ours include Support Vector Regression (SVR) [29], Gaussian KARMA [6], tree based algorithms (e.g., XGBoost (XGB) [25,26,30], Random Forest (RF) [31], and Decision Tree (DT) [32]), and neural network based algorithms (e.g., Deep Neural Networks (NN) [5], Recurrent Neural networks with simple recurrent units (RNN) [33], Long-Short Term Memory units (LSTM) [34], Gated Recurrent Units (GRU) [35], and F-DRNN [7]). (See supplementary material).

We first present single-day performance of DRNN, which is a common practice in the field [7,3,36] and is applicable when the training data is limited to a single day. Moreover, there are aspects that differ between single- and multi-day decoding, which have not yet been well characterized (e.g., varying sources of signal instability) and remain challenging in neuroscience. Furthermore, single-day decoding is important before considering multi-day decoding since our implantable hardware will be developed such that the decoder parameters can be updated at any time.

4.1 Single-day performance

We select the MWT as the input neural feature. The models are trained on the first 90% of a day and tested on the remaining 10%. Figure 2 shows the average performance of the decoders. History-Less DRNN (HL-DRNN) uses the neural data at time k and kinematics at time k-1 to make predictions at time k. As we see, DRNN and HL-DRNN are more stable and have higher average performance.
Then, we evaluate the stability and performance of all the decoders over time. Figure 31 shows that to evaluate the effect of the selected feature on the stability and performance of the DRNN, we train the DRNN on the data from the first 20 days of 2015 and test it on the consecutive days by using different features. Figure 6 shows that the DRNN operating on the MWT results in superior performance compared to the other features. Black vertical lines show the year change. We show that the wavelet features are also the best for a range of decoders in supplementary material.

4.2 Multi-day performance

To evaluate the effect of the selected feature on the stability and performance of the DRNN, we train the DRNN on the data from the first 20 days of 2015 and test it on the consecutive days by using different features. Figure 6 shows that the DRNN operating on the MWT results in superior performance compared to the other features. Black vertical lines show the year change. We show that the wavelet features are also the best for a range of decoders in supplementary material.

Then, we evaluate the stability and performance of all the decoders over time. Figure 31 shows that...
To assess the sensitivity of the decoders to the number of training days, we change the number of training days from 1 to 20 by starting from day 20. Figure 8 shows that the Deep-DRNN with 2 layers and the DRNN have higher performance compared to the other decoders, even by using a small number of training days. Moreover, figure 8 shows that the performance of the DRNN with 1 layer, 10 nodes and history of 10 is comparable to the Deep-DRNN with 2 layers, 50 and 25 nodes in the first and second layers, and history of 20. Therefore, a small DRNN with a short history has superior performance compared to the other decoders.

To evaluate the effect of re-training the DRNN, we consider four scenarios. First, we train DRNN on the first 20 days of 2015 and test it on the subsequent days. Second, we re-train a DRNN, which has been trained on 20 days, with the first 5%, 10%, 50%, and 90% of the subsequent test days. Third, we re-train the trained DRNN annually with 5%, 10%, 50%, and 90% of the first days of 2016, 2017, and 2018. Finally, we train DRNN only on the first 5% and 90% of the single test day. Figure 9 shows a general increase in the performance of the DRNN after the network is re-trained. The differences between the performances of the first three scenarios are small, which means that the DRNN does not necessarily need to be re-trained to perform well over multiple days. However, because of instability
Figure 8: Effect of number of training days on the performance of the decoders.

Figure 9: The DRNN operating in different training scenarios.

Figure 10: (a) DRNN predictions for sample targets in all four quadrants, (b) DRNN predictions - no/short neural data. True target motion (black) and reconstructions (colored)

of the recorded neural data [13], training the DRNN on the first 90% of the test day in the last scenario results in the highest average performance on the test day.

The DRNN relies on neural data inputs--not just the kinematic feedback or target information--based on the following evidence. First, target information is not explicitly provided to the DRNN. Any target information available to the DRNN is learned from the neural data and/or feedback components. Second, DRNN outputs change substantially based on different feature engineering approaches (Figures 27, 6) and over different trials (with the same features) (Figures 4, 47a). Finally, predictions fail when the DRNN uses only feedback (Feedback-Only), feedback with noise substituted for neural data (Feedback-Noise), or feedback with the neural data provided only at the beginning of the trials (Short-Neural) (Figure 47b).

5 Hardware implementation potential

BMIs are intended to operate as wireless, implantable systems that require low-power circuits, small physical size, wireless power delivery, and low temperature deltas (≤ 1°C) [37, 26, 38]. By choosing efficient algorithms that map well to CMOS technologies, Application Specific Integrated Circuit (ASIC) implementations could offer substantial power and mobility benefits. We are proposing
a method that will not only have good performance on single- and multi-day data, but will also be optimal for hardware implementation. Since it is impractical to require powerful CPUs and GPUs for everyday usage of a BMI device, we need a device that is easily portable and does not require communication of the complete signals recorded by electrodes to an external computer for computation. Doing the computation in an ASIC would drastically reduce the latency of kinematics inference and eliminate a large power draw for the gigabytes of neural data that must be transferred otherwise. Thus, we plan to create an ASIC that can be implanted in the brain to perform inference of kinematics from neural signals. The main bottleneck in most neural network accelerators is the resources spent on fetching input history and weights from memory to the Multiplication and Accumulation (MAC) unit \[39\]. The DRNN will help mitigate this issue since we require fewer nodes and input history compared to the standard recurrent neural networks. This eliminates the need for large input history storage and retrieval, reducing latency and control logic. Furthermore, by using 16-bit fixed point values for the weights and inputs rather than floating point values, we can reduce the power used by the off-chip memory \[39, 40\].

6 Discussion

We propose a Deep Multi-State DRNN with feedback and scheduled sampling to better model the nonlinearity between the neural data and kinematics in BMI applications. We show that feeding back the DRNN output recurrently result in better performance/more robust decodes. Feeding the output back to the input recurrently in addition to the input neural data provides more information to the DRNN to make predictions, which results in a smaller network with less history. Analogous to the gain term of the Kalman filter, the DRNN learns the relative importance of the neural data and feedback. Integrating both state and neural information in this way leads to smoother predictions (Figure 4). In addition, we show that the added internal derivative state enables our DRNN to track first order and more complicated patterns in the data. Our DRNN is unique since it learns a matrix that establishes a relationship between the past and present derivative states unlike the conventional DRNN. Also our DRNN, which learns all the model parameters by using back propagation through time (BPTT), is distinct from F-DRNN as the most similar previous model in BMI, which only learns the output weight by using RLS algorithm. Moreover, its application differs from most of the existing decoders that have been applied to motor cortex data of a non-human primate. To the best of our knowledge, we present the first demonstration of applying feedback and scheduled sampling to a DRNN and comparing different learning based decoders operating on different features to predict kinematics by using open-loop neural data recorded from the PPC area of a human subject in a real BMI setting. Our DRNN has the potential to be applied to the recorded data from other brain areas as a recurrent network.

To evaluate our DRNN, we analyze single-day, cross-day, and multi-day behavior of the DRNN by extracting 12 different features. Moreover, we compare the performance and robustness of the DRNN with other linear and nonlinear decoders over 43 days. Results indicate that our proposed DRNN, as a nonlinear dynamical model operating on the MWT, is a powerful candidate for a robust BMI.

The focus of this work is to first evaluate different decoders by using open-loop data since the data presented was recorded from a subject who has completed participation in the clinical trial and has had the electrodes explanted. However, the principles learned from this analysis will be relevant to the future subjects with electrodes in the same cortical area.

Future studies will evaluate the DRNN performance in a closed-loop BMI, in which all the decoders use the brain’s feedback. Next, since we believe that our small DRNN achieves higher efficiency and uses less memory by reducing the history of the input, number of weights, and therefore memory accesses, we are planning to implement the DRNN in a field-programmable gate array (FPGA) system where we can optimize for speed, area, and power usage. Then, we will build an ASIC of the DRNN for BMI applications. The system implemented must be optimized for real-time processing. The hardware will involve designing multiply-accumulates with localized memory to reduce the power consumption associated with memory fetch and memory store.

Acknowledgment: We thank Tianqiao and Chrissy (T&C) Chen Institute for Neuroscience at California Institute of Technology (Caltech) for supporting this IRB approved research. We also thank Dr. Erin Burkett for reviewing this manuscript.
References


A Dynamic recurrent neural network

A general structure of a discrete-time DRNN is given by the following expressions:

\[
\begin{align*}
s_k &= -as_{k-1} + f(W_{ss}, s_{k-1}, W_{su}, u_k, b_s) \\
\hat{y}_k &= W_{ys} s_k + b_y
\end{align*}
\]

where \( s \in \mathbb{R}^N \), \( \hat{y} \in \mathbb{R}^M \), and \( u \in \mathbb{R}^I \) are the state, prediction, and the input vectors, respectively, \( W_{ss} \in \mathbb{R}^{N \times N} \), \( W_{su} \in \mathbb{R}^{N \times I} \), and \( W_{ys} \in \mathbb{R}^{M \times N} \) are the weight matrices, \( a \in [-1, 1] \) is a constant controlling state decaying, \( b_s \in \mathbb{R}^N \), and \( b_y \in \mathbb{R}^M \) are the biases, and \( f : \mathbb{R}^N \times \mathbb{R}^I \rightarrow \mathbb{R}^N \) is a vector-valued function. [19]

B Approximation of state-space trajectories

Theorem 2.1 verifies the approximation capability of DRNNs for the discrete-time, and non-linear systems.

Theorem B.1 Let \( S \subset \mathbb{R}^M \) and \( U \subset \mathbb{R}^I \) be open sets, \( D_s \subset S \) and \( D_u \subset U \) be compact sets, and \( f : S \times U \rightarrow \mathbb{R}^M \) be a continuous vector-valued function which defines the following non-linear system

\[
z_k = f(z_{k-1}, u_k), \quad z \in \mathbb{R}^M, \quad u \in \mathbb{R}^I
\]

with an initial value \( z_0 \in D_s \). Then for an arbitrary number \( \epsilon > 0 \), and an integer \( 0 < L < \infty \), there exist an integer \( N \) and a DRNN of the form (1) with an appropriate initial state \( s_0 \) such that for any bounded input \( u : \mathbb{R}^+ = [0, +\infty) \rightarrow D_u \)

\[
\max_{0 \leq k \leq L} ||z_k - s_k|| < \epsilon
\]

Proof: See [19]

C Local stability and convergence of DRNNs

Learning rate \((\gamma)\) plays the main role in stability and convergence of neural networks. By using Lyapunov theorem, we define the range of the learning rate to guarantee the real-time convergence of DRNNs and the stability of the system during the whole control process.

Theorem C.1 If an input series of internal dynamic neural network can be activated in the whole control process subject to \( u_k \in \mathbb{R}^I \), then learning rate satisfies

\[
0 < \gamma < \frac{2}{r^2}
\]

where \( r = \frac{\partial \epsilon}{\partial W} \), \( \epsilon = \hat{y} - y \) is the difference of prediction and ground-truth, and \( W \) is the concatenation of connection weights of each network unit. Then (4) ensures the system is exponentially convergent.

Proof: See [41]

D Description of the other methods

Since all of these methods are well-known in the literature, we only provide a brief explanation of each here. We explain the F-DRNN with details since our network is a generalization of the F-DRNN, with all the parameters to be learnable. For more information, please take a look at the main references. We use Pytorch, Keras, Scikit-learn and Python 2.7 for simulations. [42, 43, 44].

D.1 Latent Factor Analysis via Dynamical Systems (LFADS) [23]

Latent Factor Analysis via Dynamical Systems (LFADS) works by modeling a dynamical system that can generate neural data. The algorithm models the nonlinear vector valued function F that
can infer firing rates using neural data input. The LFADS system is a generalization of variational auto-encoders that can be used with sequences of data, to model the time-varying aspect of neural signals. We use observed spikes as the input to the encoder RNN. We bin our spikes in 50 ms bins and then separate each center-out task into a separate trial. We use the inferred firing rates that are the result of applying a nonlinearity and affine transformation on the factors output from the generator RNN. A dimensionality of 64 was chosen for the latent variables that are the controller outputs and the factors.

D.2 FORCE Dynamic Recurrent Neural Network (F-DRNN) [7]

F-DRNN is defined as below:

\[
\begin{aligned}
\tau \frac{ds_i}{dt} &= -s_{i-1} + gW_{sr}r_{t-1} + \beta W_{si}u_t + W_{sf}\hat{y}_{t-1} + b_s \\
\hat{y}_t &= W_{yr}r_t + b_y \\
\end{aligned}
\]

where \(s \in \mathbb{R}^N\) is the activation variable, and \(r \in \mathbb{R}^N\) is the vector of corresponding firing rates. These states track the first and zero order differential features of the system, respectively. \(W_{sr} \in \mathbb{R}^{N \times N}\) describes the relationship between \(s\) and \(r\). \(W_{su} \in \mathbb{R}^{N \times I}\) relates \(s\) to the input vector \(u\). \(\hat{y}\) models the feedback in the network. \(W_{sf} \in \mathbb{R}^{N \times M}\) tracks the effect of \(\hat{y}\) on \(s\). \(W_{yr} \in \mathbb{R}^{M \times N}\) indicates the linear transformation between the firing rates \(r\) and the prediction \(\hat{y}\). \(\tau\), \(g\), and \(\beta\) are the neuronal time constant, scaling of internal connections, and scaling of inputs, respectively.

To discretize the first equation of the continuous F-DRNN, we integrate the first expression of the system by using the Euler method at a time step of \(\Delta t\). Then, the equations of the network become:

\[
\begin{aligned}
s_k &= (1-c)s_{k-1} + cgW_{sr}r_{k-1} + c\beta W_{si}u_k + cW_{sf}\hat{y}_{k-1} + cb_s \\
r_k &= \tanh(s_k) \\
\hat{y}_k &= W_{yr}r_k + b_y \\
\end{aligned}
\]

where \(c = \frac{\Delta t}{\tau}\)

In F-DRNN, we select \(W_{sr}, W_{si}\), and \(W_{sf}\) to be randomly sparse, i.e., only \(n = 0.1N, i = 0.1I\), and \(m = 0.5M\) randomly chosen elements are non-zero in each of their rows, respectively. The non-zero elements of the matrices are drawn independently from zero-mean Gaussian distributions with variances \(\frac{1}{n^2}, \frac{1}{i^2},\) and \(\frac{1}{m^2}\), respectively. \(s_0\) and the constant bias \(b_s\) are drawn from zero-mean Gaussian distributions with the standard deviations \(\sigma_s\) and \(\sigma_b\), respectively. Elements of \(W_{yr}\) are initialized to zero. Since the only matrix learned in the network is the output weight \(W_{yr}\) and all the other weights are fixed, we name this network as FORCE DRNN [7]. Therefore, the network dynamics are controlled by the matrix \(gW_{sr} + W_{sf}W_{yr}\). To update \(W_{yr}\), we use recursive least-squares (RLS) algorithm [45]. The error signal is defined by:

\[
e_k = \hat{y}_k - y_k
\]

By defining \(P\) as the inverse correlation matrix, the equation for updating \(P\) is as below:

\[
P_k = P_{k-\Delta k} - \frac{P_{k-\Delta k}r_k r_k^T r_k P_{k-\Delta k}}{1 + r_k^T P_{k-\Delta k} r_k}, \quad P_0 = \gamma I
\]

Where \(\gamma\) and \(\Delta k\) indicate learning rate and learning step size (batch size), respectively. The modification to the \(m^{th}\) column of \(W_{yr}\) by using the \(m^{th}\) element of the error is:

\[
W_{yr}^{(m)} = W_{yr}^{(m)} - e_k^{(m)} P_k r_k
\]

D.3 Deep Neural Network (NN) [5]

In a fully connected neural network, there are multiple layers: an input layer, output layer, and any number of hidden layers with multiple nodes in each hidden layer. The output of each node in each layer is connected to the input of each node in the consecutive layer. Each node performs of \(\sum_{i=1}^{N} W_i x_i\), where \(x_i\) is each input from the nodes in the previous layer and \(W_i\) is the weight of the connection between the node in the previous layer and this current node. The output is then converted
to a normalized range using a function such as $tanh$ to get values between -1 and 1. $W$ is trained through a process called back-propagation that trains the network on the inputs and finds the error, iteratively minimizing the loss function until the error stays relatively constant.

Since over-fitting is possible, which can cause issues where the trained model cannot later generalize to the separate test data, we can try to perform early stopping during validation such that a limited number of epochs (round of training with all inputs) are used for training before the weights are finalized. The following number of epochs are considered in our work: 5, 10, 20, 30, 50, 75, 100, 125, 150, 200, 300, 400, 500, 600. In addition, we consider different network structures with up to 3 layers, where each set consists of 1, 2, or 3 hidden layers with the given number of nodes in each layer: (100), (100, 100), (100, 10), (20, 20), (20, 20, 20), (40, 40), (40, 10), (40, 40, 40), (10, 10, 10).

### D.4 Support Vector Regression (SVR) [29]

Support vector regression (SVR) is the continuous form of support vector machines where the generalized error is minimized, given by the function:

$$
\hat{y} = \sum_{i=1}^{N} (\alpha_i^* - \alpha_i) k(u_i, u) + b
$$

and $\alpha_i$ are Lagrange multipliers and $k$ is a kernel function, where we use the radial basis function kernel in this paper. The Lagrange multipliers are found by minimizing a regularized risk function:

$$
\frac{1}{2} ||w||^2 + C \sum_{i=1}^{t} L_\epsilon(y) \tag{13}
$$

We vary the penalty portion of the error term, $C$, as part of the validation process to find the optimum parameter.

### D.5 Linear Model (LM) [2]

The linear model uses a standard linear regression model where we can predict kinematics ($\hat{y}$) from the neural data ($u$) by using:

$$
\hat{y} = a + \sum_{i=1}^{N} W_i u_i \tag{14}
$$

We find the weights $W_i$ and the bias term $a$ through a least squares error optimization to minimize mean squared error between the model’s predictions and true values during training. The parameters are then used to predict new kinematics data given neural data.

### D.6 Kernel Auto-Regressive Moving Average (KARMA) [6]

The Kernel Auto-Regressive Moving Average (KARMA) model can also be used for prediction. ARMA (non-kernelized) uses the following model, where $\hat{y}_k^i$ is the $i^{th}$ component of the kinematics data at time step $k$ and $u_s^j$ is the $j^{th}$ component of the neural data at time step $s$:

$$
\hat{y}_k^i = \sum_{l=1}^{r} A_l \hat{y}_{k-l}^i + \sum_{l=1}^{s} B_l u_{k-l+1}^i + e_k^i \tag{15}
$$

Thus, we are performing a weighted average of the past $r$ time steps of kinematics data and the past $s$ time steps of neural data (as well as the current one) with a residual error term, $e$. Then, the difference in KARMA is that we use the kernel method to translate data to the radial basis function dimension. We use a standard SVR solver for inference, by just concatenating the different histories for the kinematics and neural data. When training, we use the known kinematics values for the history. However, when predicting new kinematics data, we use old predictions for the history portion of the new predictions.
D.7 Kalman Filter (KF) [3]

The Kalman Filter combines the idea that kinematics are a function of neural firings as well as the idea that neural activity is a function of movements, or the kinematics. This can be represented by two equations:

\[
\begin{align*}
\dot{\hat{y}}_{k+1} &= A_k \hat{y}_k + w_k \\
u_k &= H_k \hat{y}_k + q_k
\end{align*}
\]  

These represent how the system evolves over time as well as how neural activity is generated by system’s behavior. The matrices \(A, H, Q, \text{ and } W\) can be found through a training process (where \(q \sim \mathcal{N}(0, Q)\) and \(w \sim \mathcal{N}(0, W)\). Using properties of the conditional probabilities of kinematics and neural data, we get a closed form solution for maximizing the joint probability \(p(Y_M, U_M)\). Using the physical properties of the problem, we get matrix \(A\) to be of the form:

\[
A = \begin{bmatrix}
1 & 0 & \Delta t & 0 \\
0 & 1 & 0 & \Delta t \\
0 & 0 & a & b \\
0 & 0 & c & d
\end{bmatrix}
\]  

We define \(A_v\) as:

\[
A_v = \begin{bmatrix}
a & b \\
c & d
\end{bmatrix} = V_2 V_1^T (V_1 V_1^T)^{-1}
\]  

\(V_1\) consists of the velocity kinematics points except for the last time step, \(V_2\) consists of the velocity kinematics points except for the first time step, and \(dt\) is the time step size used, 50 ms in our case.

Furthermore \(W\) is a zero matrix with the matrix \(W_v = \frac{1}{N-1} (V_2 - AV_1)(V_2 - AV_1)^T\) in the bottom right corner. \(H\) and \(Q\) are given by:

\[
\begin{align*}
H &= U^T Y (YY^T)^{-1} \\
Q &= \frac{1}{N} (U - HY)(U - HY)^{-1}
\end{align*}
\]

Then, we can use the update equations:

\[
\begin{align*}
\dot{\hat{y}}_k &= A \hat{y}_{k-1} \\
P_k^- &= A P_{k-1}^A + W \\
\hat{y}_k &= \hat{y}_k^- + K_k (u_k - H \hat{y}_k^-) \\
P_k &= (1 - K_k H) P_k^-
\end{align*}
\]  

Here, \(P\) is the covariance matrix of the kinematics. \(K_k\), the Kalman filter gain is given by:

\[
K_k = P_k^H (H P_k^- H^T + Q)^{-1}
\]

D.8 Recurrent Neural Network (RNN) [33]

A vanilla recurrent neural network with \(N\) hidden nodes for regression is defined as:

\[
\begin{align*}
r_k &= \tanh(W_r r_{k-1} + W_i u_k + b_r) \\
y_k &= W_y r_k + b_y
\end{align*}
\]  

where \(r \in \mathbb{R}^N, \hat{y} \in \mathbb{R}^M\), and \(u \in \mathbb{R}^I\) are the state, prediction, and input vectors, respectively, \(W_r \in \mathbb{R}^{N \times N}, W_{ru} \in \mathbb{R}^{N \times I}, \text{ and } W_y \in \mathbb{R}^{M \times N}\) are the weight matrices, \(b_r \in \mathbb{R}^N\) and \(b_y \in \mathbb{R}^M\) are the biases.

Because of the internal state \(r\) which acts as a history unit, the RNN is capable of remembering and extracting short term temporal dependencies in sequential data. Therefore, to find the spatio-temporal relationship between the recorded neural data and kinematics as sequential data, we train an RNN with optimal parameters and compare its performance with the DRNN.
D.9 Long-Short Term Memory Recurrent Neural Network (LSTM) [34]

It is well-known that Simple RNN units cannot remember long term dependencies in sequential data because of the vanishing gradients problem. Another version of RNNs that is widely used in the literature are RNNs with Long-Short Term Memory (LSTM) units. By denoting \( \odot \) as Hadamard product, the LSTM is defined as:

\[
\begin{aligned}
  f_k &= \sigma(W_{fu}u_k + W_{fr}r_{k-1} + b_f) \\
  i_k &= \sigma(W_{iu}u_k + W_{ir}r_{k-1} + b_i) \\
  o_k &= \sigma(W_{ou}u_k + W_{or}r_{k-1} + b_o) \\
  c_u &= \tanh(W_{cu}u_k + W_{cr}r_{k-1} + b_c) \\
  c_k &= f_k \odot c_{k-1} + i_k \odot c_u \\
  r_k &= o_k \odot \tanh(c_k) \\
  y_k &= W_yr_k + b_y
\end{aligned}
\]  \\

(23)

\( r_k \) is the hidden state as in Simple RNN, \( c_u \) is the output from the cell update activation function, \( c_k \) is the LSTM cell’s internal state, \( f_k, i_k, o_k \) are the output matrices from the respective forget, input, and output activation functions, which act as the LSTM’s gates, \( W \) and \( b \) represent the weights and biases, and \( \sigma \) is the sigmoid function.

D.10 Gated Recurrent Units (GRU) [35]

A simpler version of the RNN cells than LSTM that can extract long term dependencies in sequential data are Gated Recurrent Units (GRU). The GRU formulation is as below:

\[
\begin{aligned}
  z_k &= \sigma(W_{zu}u_k + W_{zr}r_{k-1} + b_z) \\
  h_k &= \sigma(W_{hu}u_k + W_{hr}r_{k-1} + b_h) \\
  r_u &= \tanh(W_{ru}u_k + W_{rr}(h_k \odot r_{k-1}) + b_r) \\
  r_k &= (1 - z_k) \odot r_{k-1} + z_k \odot r_u \\
  y_k &= W_yr_k + b_y
\end{aligned}
\]  \\

(24)

Here, \( h \) is a reset gate, and \( z \) is an update gate. The reset gate determines how to combine the previous memory and the new input. The network decides how much of the previous memory should be kept by using the update gate. Vanilla RNN is the case that we set the update gate to all 0’s and the reset to all 1’s.

D.11 XGBoost (XGB) [25][26]

XGBoost is one kind of boosting methods which uses ensemble of decision trees. Among 29 competitions winning solutions published at Kaggle during 2015, 17 solutions used XGBoost [25]. For a given data set with \( n \) examples and \( m \) features \( D = \{(x_i, y_i)\}, |D| = n, x_i \in \mathbb{R}^m, y_i \in \mathbb{R}, \) a tree ensemble model uses \( K \) additive functions to predict the output:

\[
\hat{y}_i = p(x_i) = \sum_{k=1}^{K} f_k(x_i), f_k \in F
\]

(25)

where \( F = \{ f(x) = w_{q(x)} \}, q : \mathbb{R}^m \rightarrow T, w \in \mathbb{R}^T \) is the space of regression trees, \( q \) represents the structure of each tree, \( T \) is the number of leaves, and each \( f_k \) corresponds to a tree structure \( q \) and leaf weights \( w \).

D.12 Random Forests (RF) and Decision Trees (DT) [31][32]

Random Forests are one kind of bagging tree based algorithms that make the prediction by routing a feature sample through the tree to the leaf randomly. The training process will be done independently for each tree. The forest final prediction is the average of the predictions of all the trees. Decision trees are a special case of random forests with one tree.
E DRNN training: back propagation through time (BPTT)

The squared loss function is defined as below:

\[ J = \frac{1}{2}(\hat{y} - y)^2 \]  

(26)

Therefore, the average loss at time step \( k \) is:

\[ J_k = \frac{1}{k} \sum_{t=1}^{k} J_t \]  

(27)

Training the network is usually accomplished by applying a mini-batch optimization method to search for a set of parameters that maximize the log-likelihood function:

\[ \theta^* = \arg \max_{\theta} \sum_{(x^i, y^i)} \log P(y^i | x^i; \theta) \]  

(28)

\((x^i, y^i)\) is a training pair, \( P \) is the probability distribution of the data, and \( \theta^* \) is the optimum set of parameters.

To update the model’s parameters, we perform back propagation through time (BPTT) by using Adam optimization algorithm \([46]\) (Supplementary materials). We need to find the following derivatives:

\[ \frac{\partial J_k}{\partial W_{ss}}, \frac{\partial J_k}{\partial W_{sr}}, \frac{\partial J_k}{\partial W_{si}}, \frac{\partial J_k}{\partial W_{sf}}, \frac{\partial J_k}{\partial b_s}, \frac{\partial J_k}{\partial W_{yr}}, \frac{\partial J_k}{\partial b_y} \]

Since the partial derivative is a linear operator, we have:

\[ \frac{\partial J_k}{\partial \theta} = \frac{1}{k} \sum_{t=1}^{k} \frac{\partial J_t}{\partial \theta} \]  

(29)

where \( \theta \) indicates weights or biases.

\[ \frac{\partial J_k}{\partial W_{yr}} \text{ and } \frac{\partial J_k}{\partial b_y} \] depend only on the variables at the present time. Therefore, by using the chain rule, we get

\[ \frac{\partial J_t}{\partial \theta} = \frac{\partial J_k}{\partial \hat{y}_k} \frac{\partial \hat{y}_k}{\partial \theta} \]  

(30)

where \( \theta \in \{W_{yr}, b_y\} \)

The derivatives of \( J \) with respect to all the other parameters depend not only on the present time variables, but also they depend on the history of the variables from the beginning of the update time (batch size). Therefore, we have:

\[ \frac{\partial J_t}{\partial \theta} = \sum_{i=0}^{t} \frac{\partial J_i}{\partial r_i} \frac{\partial r_i}{\partial s_i} \frac{\partial s_i}{\partial \theta} \]  

(31)

where \( \theta \in \{W_{ss}, W_{sr}, W_{si}, W_{sf}, b_s\} \)

BPTT has been implemented by using Pytorch automatic differentiation package \([44]\).

F Adam optimization method \([46]\)

Adam optimization algorithm is widely used for parameter optimization in neural network based learning methods. The algorithm is as below:

G Performance evaluation measures

As a pre-processing step before passing the neural data to the decoders, we use XGBoost feature importance score to select stable channels across the training days. The more a feature is used to
Algorithm 2 Adam Stochastic Optimization Algorithm

Require: \( \gamma \): Learning rate
Require: \( \beta_1, \beta_2 \in [0, 1) \): Exponential decay rates for moment estimates
Require: \( J(\theta) \): Loss function with parameter \( \theta \)
Require \( \theta_0 \): Initial parameter vector, \( m_0 \leftarrow 0 \) (Initialize 1st moment vector), \( v_0 \leftarrow 0 \) (Initialize 2nd moment vector), \( t \leftarrow 0 \) (Initialize timestep)

while \( \theta_t \) not converged do
  \( t \leftarrow t + 1 \)
  \( g_t \leftarrow \nabla J(\theta_{t-1}) \) (Get gradients w.r.t. stochastic objective at timestep \( t \))
  \( m_t \leftarrow \beta_1 m_{t-1} + (1 - \beta_1) g_t \) (Update biased 1st moment estimate)
  \( v_t \leftarrow \beta_2 v_{t-1} + (1 - \beta_2) g_t^2 \) (Update biased 2nd raw moment estimate)
  \( \hat{m}_t \leftarrow \frac{m_t}{(1 - \beta_1^t)} \) (Compute bias-corrected 1st moment estimate)
  \( \hat{v}_t \leftarrow \frac{v_t}{(1 - \beta_2^t)} \) (Compute bias-corrected 2nd raw moment estimate)
  \( \theta_t \leftarrow \theta_{t-1} - \gamma \frac{\hat{m}_t}{\sqrt{\hat{v}_t + \epsilon}} \) (Update parameters)
end while

return \( \theta_t \) (Resulting parameters)

make key decisions with XGBoost decision trees, the higher its relative importance. This importance is calculated explicitly for each feature in the dataset, allowing features to be ranked and compared to each other. Importance is calculated for a single decision tree by the amount that each feature split point improves the performance measure, weighted by the number of observations the node is responsible for. The importances are then averaged across all of the XGBoost decision trees.

To compare the decoders’ predictions, we report Root Mean Square Error (RMSE), and \( R^2 \). The RMSE finds the average point-wise error between the predicted and ground-truth signals as below:

\[
RMSE = \sqrt{\frac{1}{K} \sum_{i=1}^{K} (y_i - \hat{y}_i)^2}
\]  

(32)

where \( K \) is the total number of data points, \( y_i \) and \( \hat{y}_i \) are the \( i^{th} \) ground-truth and prediction, respectively. The smaller the RMSE is, the better the performance.

The \( R^2 \) measures the strength of the linear association between the predicted and the ground-truth signals as below:

\[
R^2 = \left( \frac{\sum_i (y_i - \bar{y})(\hat{y}_i - \bar{\hat{y}})}{\sqrt{\sum_i (y_i - \bar{y})^2 \sum_i (\hat{y}_i - \bar{\hat{y}})^2}} \right)^2
\]  

(33)

that is a real number varies from 0 to 1. The larger the \( R^2 \) is, the better the performance.

H Experimental results for other kinematics

Table 2 shows the parameters of all the decoders for single- and multi-day analysis. The average performance of a small DRNN with a short history surpasses the other decoders’ in terms of \( R^2 \) and RMSE. Even the HL-DRNN’s results are comparable with the other larger recurrent models with longer histories. Figures 11, 12, and 13 show the single-day predictions of all the decoders on a sample day for \( V_y, X \), and \( V_x \), respectively. Figures 14 and 15 show the average performance of all the decoders.
Table 2: Optimum parameters for different algorithms (Only differences are reported for multi-day)

<table>
<thead>
<tr>
<th>MODEL</th>
<th>SINGLE-DAY</th>
<th>MULTI-DAY</th>
</tr>
</thead>
<tbody>
<tr>
<td>SVR</td>
<td>$C : 0.1$, Kernel: RBF</td>
<td>$C : 1$</td>
</tr>
<tr>
<td>KARMA</td>
<td>$r : 0, s : 20, C : 0.1$, Kernel: Gaussian</td>
<td>$r : 0, s : 2, C : 0.1$</td>
</tr>
<tr>
<td>XGB</td>
<td>number of trees: 15, maximum depth: 8</td>
<td>number of trees: 20</td>
</tr>
<tr>
<td>RF</td>
<td>number of trees: 20, maximum depth: 10</td>
<td>number of trees: 40</td>
</tr>
<tr>
<td>DT</td>
<td>maximum depth: 10</td>
<td>-</td>
</tr>
<tr>
<td>NN</td>
<td>Layer: 2, Optimizer: Adam, Nodes: (40, 10), Batch size: 64, dropout: 0, epoch: 118</td>
<td>Batch size: 128, dropout: 0.25</td>
</tr>
<tr>
<td>RNN</td>
<td>Optimizer: RMSprop, Nodes: 25, Batch size: 64, History: 20, dropout: 0.2, epoch: 19</td>
<td>Nodes: 100, Batch size: 128, History: 40, epoch: 50</td>
</tr>
<tr>
<td>LSTM</td>
<td>Optimizer: RMSprop, Nodes: 50, Batch size: 64, History: 40, dropout: 0.35, epoch: 17</td>
<td>Nodes: 75, Batch size: 128, epoch: 50</td>
</tr>
<tr>
<td>GRU</td>
<td>Optimizer: RMSprop, Nodes: 75, Batch size: 32, History: 40, dropout: 0.3, epoch: 18</td>
<td>Batch size: 64</td>
</tr>
<tr>
<td>FDRNN</td>
<td>$g : 1, \beta : 0.5$, Nodes: 1200, Batch size: 10, $\sigma_b : 0.025, \sigma_s : 0.01, \tau : 250$ ms, epoch: 10</td>
<td>$g : 0.5$, Nodes: 1500</td>
</tr>
<tr>
<td>DRNN</td>
<td>Layer: 1, Optimizer: Adam, Nodes: 5, $p_s : 0.25, p_f : 0$ Batch size: 16, History: 10, dropout: 0.25, epoch: 2</td>
<td>Nodes: 10, $p_s : 0.5$ Batch size: 64, epoch: 5</td>
</tr>
</tbody>
</table>

We do three more single-day analyses. First, we evaluate the effect of neural data history on the recurrent networks. Figures 16 and 17 show the performance of recurrent decoders versus history of neural data. The performance of other recurrent decoders drops, however, the DRNN’s performance is stable. Second, we evaluate the performance of the recurrent networks with different number of internal hidden nodes. Figures 18 and 19 show the performance of these decoders versus the number of hidden nodes. The DRNN’s performance has little changes when we add more nodes to the network. Moreover, it still performs superior to the other recurrent decoders. Third, we assess the stability of the recurrent networks to the amount of the single day’s training data. Figures 20 and 21 show that the DRNN’s performance is still better than the others. Moreover, it implies that the DRNN is a stable decoder and it works well even with less than 10% single-day training data.

Since we have only presented the $R^2$ values for Y-axis position in the body of the paper, we first present the corresponding RMSE values in figures 22, 23, 24, 25, and 26. These figures evaluate the average cross-day performance of the decoders by operating on different features, average multi-day performance of the DRNN on different features, average multi-day performance of the decoders operating on the MWT, effect of the used number of training days on the average performance of the decoders, and the effect of re-training on the DRNN’s long term stability and performance in different scenarios, respectively.

Subsequent figures present the equivalent $R^2$ and RMSE values for all the other kinematics. Figures 27 and 28 show the cross-day performance of the decoders. Figures 29 and 30 show the multi-day performance of the DRNN on the different features. Figures 31 and 32 show the performance of all the decoders operating on the MWT. Figures 25, 33, 35, 37, 39, 41, and 43 show the effect of number of training days on the performance of all the decoders. Finally, figures 26, 34, 36, 38, 40, 42, and 44 show the effect of re-training on the DRNN’s long term stability and performance in the different scenarios.
Figure 11: $V_y$ velocity regression of different algorithms on test data from the same day 2018-04-23: true target motion (black) and reconstruction (red).

Figure 12: $X$ position regression of different algorithms on test data from the same day 2018-04-23: true target motion (black) and reconstruction (red).
Figure 13: Vx velocity regression of different algorithms on test data from the same day 2018-04-23: true target motion (black) and reconstruction (red).

Figure 14: Average performance of the decoders operating on the MWT over single-day data.
Figure 15: Average RMSE of decoders operating on the MWT over single-day data.
Figure 16: Performance of the recurrent networks versus the amount of single-day history of neural data.
Figure 17: RMSE of the recurrent networks versus the amount of single-day history of neural data.
Figure 18: Performance of the recurrent networks on single-day data versus number of internal hidden nodes of the networks.
Figure 19: RMSE of the recurrent networks on single-day data versus number of internal hidden nodes of the networks.
Figure 20: The effect of the amount of single-day training data on the performance of the recurrent networks.
Figure 21: The effect of the amount of single-day training data on the performance of the recurrent networks.
Figure 22: Cross-day analysis of the DRNN.

Figure 23: The DRNN operating on different features.
Figure 24: Multi-day performance of the decoders.

(a) Y

(b) Y

Figure 25: Effect of number of training days on the performance of decoders - Y.

Figure 26: The DRNN operating in different training scenarios - Y.
Figure 27: Cross-day analysis of the DRNN.
Figure 28: Cross-day analysis of the DRNN.
Figure 29: The DRNN operating on different features.
Figure 30: The DRNN operating on different features.
Figure 31: Multi-day performance of the decoders.
Figure 32: Multi-day performance of the decoders.
Figure 39: Effect of number of training days on the performance of decoders - X.

Figure 40: The DRNN operating in different training scenarios - X.

Figure 41: Effect of number of training days on the performance of decoders - Vx.

Figure 42: The DRNN operating in different training scenarios - Vx.

Figure 43: Effect of number of training days on the performance of decoders - Vx.

Figure 44: The DRNN operating in different training scenarios - Vx.
Figure 45: (a) DRNN predictions for sample targets in all four quadrants, (b) DRNN predictions - no/short neural data. True target motion (black) and reconstructions (colored) - Vy

Figure 46: (a) DRNN predictions for sample targets in all four quadrants, (b) DRNN predictions - no/short neural data. True target motion (black) and reconstructions (colored) - X

Figure 47: (a) DRNN predictions for sample targets in all four quadrants, (b) DRNN predictions - no/short neural data. True target motion (black) and reconstructions (colored) - Vx
I Performance of all the decoders operating on different features

I.1 Long-Short Term Memory Recurrent Neural Network (LSTM)

Figure 48: The LSTM operating on different features
Figure 49: The LSTM operating on different features
1.2 Gated Recurrent Units (GRU)

Figure 50: The GRU operating on different features
Figure 51: The GRU operating on different features
1.3 Recurrent Neural Network (RNN)

Figure 52: The RNN operating on different features
Figure 53: The RNN operating on different features
1.4 FORCE Dynamic Recurrent Neural Network (F-DRNN)

![Graphs showing R^2 values over consecutive days for different features](image)

(a) Y  (b) Y
(c) Vy  (d) Vy
(e) X  (f) X
(g) Vx  (h) Vx

Figure 54: The F-DRNN operating on different features
Figure 55: The F-DRNN operating on different features
1.5 Deep Neural Network (NN)

Figure 56: The NN operating on different features
Figure 57: The NN operating on different features
1.6 Support Vector Regression (SVR)

Figure 58: The SVR operating on different features
Figure 59: The SVR operating on different features
1.7 XGBoost (XGB)

Figure 60: The XGB operating on different features
Figure 61: The XGB operating on different features
1.8 Random Forests (RF)

Figure 62: The RF operating on different features
Figure 63: The RF operating on different features
1.9 Decision Tree (DT)

Figure 64: The DT operating on different features
Figure 65: The DT operating on different features
I.10 Kernel Auto-Regressive Moving Average (KARMA)

Figure 66: The KARMA operating on different features
Figure 67: The KARMA operating on different features
1.11 Kalman Filter (KF)

Figure 68: The KF operating on different features
Figure 69: The KF operating on different features
I.12 Linear Model (LM)

Figure 70: The LM operating on different features
Figure 71: The LM operating on different features