

Method	Equations, details, and hyperparameters
<i>General information</i>	<p>For all the methods below, we describe <i>our particular uses</i>, which are available in our code package and are used in the decoding demonstrations we show in <i>Results</i>.</p> <p>Let <math>\mathbf{X}</math> be the input matrix of covariates and let <math>\mathbf{Y}</math> be the output that is being predicted. <math>\mathbf{Y}</math> is either a vector or matrix depending on whether the outputs (e.g. <math>x</math> and <math>y</math> components of velocity) are predicted simultaneously.</p> <p>Also, note that for our demonstrations, for all methods below, the noise of the output is assumed to be normally distributed.</p>
Wiener Filter	<p>Decoder predictions have the form <math>\mathbf{Y} = \boldsymbol{\theta}\mathbf{X}</math>, where <math>\boldsymbol{\theta}</math> is fit according to maximum likelihood estimation.</p> <p><i>Hyperparameters:</i> None</p>
Wiener Cascade	<p>Decoder predictions have the form <math>\mathbf{Y} = \beta_0 + \beta_1\mathbf{Z}^1 + \dots + \beta_n\mathbf{Z}^n</math>, where <math>\mathbf{Z} = \boldsymbol{\theta}\mathbf{X}</math>. The parameters <math>\boldsymbol{\theta}</math> and the <math>\beta</math>'s are fit iteratively according to maximum likelihood estimation.</p> <p><i>Hyperparameters:</i> Degree of the polynomial used for the nonlinearity, <math>n</math>.</p>
SVR	<p>Tutorials have previously been written on the details of support vector regression and how they are optimized (Smola and Schölkopf, 2004). Here, we provide a very short mathematical summary. In nonlinear SVR:</p> $\mathbf{Y} = \sum_{i=1}^l (\alpha_i - \alpha_i^*)k(\mathbf{X}_i, \mathbf{X}) + b$ <p>where <math>i</math> represents the data point. <math>k</math> is the kernel function, which allows for the nonlinear transformation. We used a radial basis function kernel:</p> $k(\mathbf{x}', \mathbf{x}) = \exp\left(-\frac{\ \mathbf{x}' - \mathbf{x}\ ^2}{\sigma^2}\right)$ <p>The parameters <math>\alpha_i</math> and <math>\alpha_i^*</math> are constrained to be in the range <math>[0, C]</math> in the optimization problem, where <math>C</math> is a hyperparameter that can be understood as being inversely related to regularization strength.</p> <p><i>Hyperparameters:</i> Penalty of the error term (<math>C</math>), maximum number of iterations</p>
XGBoost	<p>In gradient-boosted tree methods (which XGBoost is an implementation of), a mapping is learned using a collection of regression trees. Training an individual regression tree is accomplished by sequentially splitting the input space, and assigning an output value to each final partition of the input space (see (Breiman, 2017) for further description). Let <math>f_i</math> be the input/output mapping of regression tree <math>i</math>. The final decoder, which combines across <math>N</math> regression trees, will have the form:</p> $\mathbf{Y} = \sum_{k=1}^N f_k(\mathbf{X})$

	<p>To train the XGBoost decoder, regression trees are trained sequentially. The first tree is trained to minimize the loss function which here is <math>\ Y - f_1X\ _2^2 + \Omega(f_1)</math>, where <math>\Omega</math> is a regularization function. The second tree is then trained to predict the residual of the first tree's predictions. That is, we optimize <math>f_2</math> to minimize <math>\ Y - f_1X - f_2X\ _2^2 + \Omega(f_2)</math>. Successive trees continue to be optimized to predict the remaining residual. See <a href="https://xgboost.readthedocs.io/en/latest/tutorials/model.html">https://xgboost.readthedocs.io/en/latest/tutorials/model.html</a> for a more in-depth tutorial description.</p> <p><i>Hyperparameters:</i> Maximum depth of each regression tree, number of trees, learning rate</p>
Feedforward Neural Net	<p>We created a fully connected (dense) feedforward neural network with 2 hidden layers and rectified linear unit (“ReLU”) (Glorot et al., 2011) activations after each hidden layer. We required the number of hidden units in each layer to be the same. This can be written as:</p> $\begin{aligned} H_1 &= \text{ReLU}(W_1X + b_1) \\ H_2 &= \text{ReLU}(W_2X + b_2) \\ Y &= W_3H_2 + b_3 \end{aligned}$ <p>We used the Adam algorithm (Kingma and Ba, 2014) as the optimization routine to solve for all <math>W</math>'s and <math>b</math>'s. We used the Keras library (Chollet, 2015) for all neural network implementations. This neural network, and all neural networks below had two output units. That is, the same network predicted the x and y components (of position or velocity) together, rather than separately.</p> <p><i>Hyperparameters:</i> Number of units (dimensionality of <math>H_1</math> and <math>H_2</math>), amount of dropout during training, number of training epochs</p>
Simple RNN	<p>Decoder predictions have the form:</p> $\begin{aligned} H_t &= \text{ReLU}(UX_t + WH_{t-1} + b) \\ Y_t &= VH_t + c \end{aligned}$ <p>Note that we used the ReLU nonlinearity as opposed to the “tanh” nonlinearity, which is more common in RNNs, as this improved performance. We used RMSprop (Tieleman and Hinton, 2012) as the optimization routine.</p> <p><i>Hyperparameters:</i> Number of units (dimensionality of <math>H</math>), amount of dropout during training, number of training epochs</p>
GRUs	<p>Here, we mix notation from (Goodfellow et al., 2016) and (Olah, 2015), which provide excellent descriptions of the method. The GRU has two gates, which control the transmission of information through the network. The “update” and “reset” gates are parameterized as follows:</p> <p>Update gate: <math>z_t = \sigma(U_zX_t + W_zH_{t-1} + b_z)</math>  Reset gate: <math>r_t = \sigma(U_rX_t + W_rH_{t-1} + b_r)</math></p> <p>A temporary hidden state is calculated as:</p>

	$\tilde{h}_t = \tanh(U_h X_t + r_t * W_h H_{t-1} + b_c)$ <p>In this calculation, the reset gate controls what information from the previous hidden state is in this proposed new hidden state. Note that the * symbol denotes an elementwise (by neural network unit) multiplication. To calculate the hidden state, we use the update gate to control what information flows from the previous hidden state versus what is updated from the temporary state:</p> $h_t = (1 - z_t) * h_{t-1} + z_t * \tilde{h}_t$ <p>Finally, as in the standard RNN:</p> $Y_t = V H_t + c$ <p>We used RMSprop (Tieleman and Hinton, 2012) as the optimization routine.</p> <p><i>Hyperparameters:</i> Number of units (dimensionality of <math>H</math>), amount of dropout during training, number of training epochs.</p>
LSTM	<p>As above, we mix notation from (Goodfellow et al., 2016) and (Olah, 2015), which provide excellent descriptions of the method. The LSTM has three gates, which control the transmission of information through the network. The “forget”, “input”, and “output” gates are parameterized as follows:</p> $f_t = \sigma(U_f X_t + W_f H_{t-1} + b_f)$ $i_t = \sigma(U_i X_t + W_i H_{t-1} + b_i)$ $o_t = \sigma(U_o X_t + W_o H_{t-1} + b_o)$ <p>The LSTM has a cell state, <math>C_t</math>, that carries information across time. We can calculate <math>C_t</math>, as:</p> $\tilde{C}_t = \tanh(U_c X_t + W_c H_{t-1} + b_c)$ $C_t = f_t * C_{t-1} + i_t * \tilde{C}_t$ <p>where <math>\tilde{C}_t</math> is a temporary cell state, that becomes part of the cell state depending on the input gate, <math>i_t</math>. Additionally, the previous time point’s cell state is carried through depending on the forget gate, <math>f_t</math>. Note that the * symbol denotes an elementwise (by neural network unit) multiplication.</p> <p>The hidden state of the network is an output-gated version of the cell state:</p> $H_t = o_t * \tanh(C_t)$ <p>Finally, as in the standard RNN:</p> $Y_t = V H_t + c$ <p>We used RMSprop (Tieleman and Hinton, 2012) as the optimization routine.</p> <p><i>Hyperparameters:</i> Number of units (dimensionality of <math>H</math>), amount of</p>

	dropout during training, number of training epochs
Kalman Filter	<p>In the standard Kalman Filter,</p> $\mathbf{y}_t = \mathbf{A}\mathbf{y}_{t-1} + \mathbf{w}$ <p>where <math>\mathbf{y}_t</math> and <math>\mathbf{y}_{t-1}</math> are 6 x 1 vectors reflecting kinematic variables (not just the velocity or position being predicting), and <math>\mathbf{w}</math> is sampled from a normal distribution, <math>N(0, \mathbf{W})</math>, with mean 0 and covariance <math>\mathbf{W}</math>.</p> $\mathbf{x}_{t^*} = \mathbf{H}\mathbf{y}_t + \mathbf{q}$ <p>where <math>\mathbf{x}_{t^*}</math> is the neural activity at time <math>t^*</math>, and <math>\mathbf{q}</math> is sampled from a normal distribution, <math>N(0, \mathbf{Q})</math>. Note that we allowed a lag between the neural data and predicted kinematics, which is why we use <math>t^*</math> for the time of the neural activity.</p> <p>During training, <math>\mathbf{A}</math>, <math>\mathbf{H}</math>, <math>\mathbf{W}</math>, and <math>\mathbf{Q}</math> are empirically fit on the training set using maximum likelihood estimation. When making predictions, to update the estimated hidden state at a given time point, the updates derived from the current measurement and the previous hidden states are combined. During this combination, the noise matrices give a higher weight to the less uncertain information. See (Wu et al., 2003) or our code for the update equations (note that <math>\mathbf{x}</math> and <math>\mathbf{y}</math> have different notation in (Wu et al., 2003)).</p> <p>We had one hyperparameter which differed from the standard implementation (Wu et al., 2003). We divided the noise matrix associated with the transition in kinematic states, <math>\mathbf{W}</math>, by the hyperparameter scalar <math>C</math>, which allowed weighting the neural evidence and kinematic transitions differently. The rationale for this addition is that neurons have temporal correlations, which make it desirable to have a parameter that allows changing the weight of the new neural evidence. The introduction of this parameter made a big difference for the hippocampus dataset (Extended Data Fig. 4 - 1).</p> <p><i>Hyperparameters:</i> <math>C</math> and the lag between the neural data and predicted kinematics.</p>
Naïve Bayes	<p>We used a Naïve Bayes decoder similar to the one implemented in (Zhang et al., 1998). We first fit an encoding model (tuning curve) using the output variables. Let <math>f_i(\mathbf{s})</math> be the value of the tuning curve (the expected number of spikes) for neuron <math>i</math> at the output variables <math>\mathbf{s}</math>. Note that <math>\mathbf{s}</math> is a vector containing the two output variables we are predicting (<math>x</math> and <math>y</math> positions/velocities). We assume the number of recorded spikes in the given bin, <math>r_i</math>, is generated from the tuning curve with Poisson statistics:</p> $P(r_i   \mathbf{s}) = \frac{\exp[-f_i(\mathbf{s})] f_i(\mathbf{s})^{r_i}}{r_i!}$ <p>We also assume that all the neurons' spike counts are conditionally independent given the output variables, so that:</p>

$$P(\mathbf{r}|\mathbf{s}) \propto \prod_i P(r_i|\mathbf{s})$$

where  $\mathbf{r}$  is a vector with the spike counts of all neurons. Bayes' rule can then be used to determine the likelihood of the output variables given the spike counts of all neurons:

$$P(\mathbf{s}|\mathbf{r}) \propto P(\mathbf{r}|\mathbf{s})P(\mathbf{s})$$

where  $P(\mathbf{s})$  is the probability distribution of the output variables. To help with temporal continuity of decoding, we want our probabilistic model to include how the output variables at one time step depend on the output variables at the previous time step:  $P(\mathbf{s}_t|\mathbf{s}_{t-1})$ . Thus, we can more generally write, using Bayes' rule as before:

$$P(\mathbf{s}_t|\mathbf{r}_{t^*}, \mathbf{s}_{t-1}) \propto P(\mathbf{r}_{t^*}|\mathbf{s}_t)P(\mathbf{s}_{t-1}|\mathbf{s}_t)P(\mathbf{s}_t)$$

Note that we use  $\mathbf{r}_{t^*}$  rather than  $\mathbf{r}_t$  because we use neural responses from multiple time bins to predict the current output variables. The above formula assumes that  $\mathbf{r}_{t^*}$  and  $\mathbf{s}_{t-1}$  are independent, conditioned on  $\mathbf{s}_t$ . The final decoded stimulus in a time bin is:  $\text{argmax}_{\mathbf{s}_t} P(\mathbf{s}_t|\mathbf{r}_{t^*}, \mathbf{s}_{t-1})$ .

$P(\mathbf{s}_{t-1}|\mathbf{s}_t)$  was determined as follows. Let  $\Delta\mathbf{s}$  be the Euclidean distance in  $\mathbf{s}$  from one time step to the next. We fit  $P(\Delta\mathbf{s})$  as a Gaussian using data from the training set.  $P(\mathbf{s}_{t-1}|\mathbf{s}_t)$  was approximated as  $P(\Delta\mathbf{s}_t)$ . That is, the probability of going from one output state to another was only based on the distance between the output states, not the output state itself.

Additionally, including  $P(\mathbf{s})$  based on the distribution of output variables in the training set did not improve performance on the validation set. This could be because the probability distribution differed between the training and validation/testing sets, or because the distribution of output variables was approximately uniform in our tasks. Thus, we simply used a uniform prior.

In our calculations, we discretize  $\mathbf{s}$  into a 100 x 100 grid going from the minimum to maximum of the output variables. When increasing the decoding resolution of the output variables, we did not see a meaningful change in decoding accuracy.

Our tuning curves had the format of a Poisson generalized quadratic model (Park et al., 2013), which improved the performance over generalized linear models on validation datasets.

On the hippocampus dataset, we used the total number of spikes over the same time interval as we used for the other decoders (4 bins before, the concurrent bin, and 5 bins after). Note that using a single time bin of spikes led to very poor performance. On the motor cortex and somatosensory

	<p>cortex datasets, the naïve Bayes decoder gave very poor performance regardless of the bins used. We ultimately used bins that gave the best performance on a validation set: 2 bins before and the concurrent bin for the motor cortex dataset; 1 bin before, the concurrent bin, and 1 bin after for the somatosensory cortex dataset.</p> <p><i>Hyperparameters: None</i></p>
--	--

**Figure 4-3: Additional decoder details, including equations and hyperparameters**

These details are for the decoder implementations that we use in our demonstrations and have in our code package.

**References**

Breiman L (2017) Classification and regression trees: Routledge.

Chollet F (2015) Keras.

Glorot X, Bordes A, Bengio Y (2011) Deep sparse rectifier neural networks. In: Proceedings of the Fourteenth International Conference on Artificial Intelligence and Statistics, pp 315-323.

Goodfellow I, Bengio Y, Courville A (2016) Deep learning: MIT press.

Kingma D, Ba J (2014) Adam: A method for stochastic optimization. arXiv preprint arXiv:1412.6980.

Olah C (2015) Understanding LSTM Networks. Available from: <https://colah.github.io/posts/2015-08-Understanding-LSTMs/>.

Park IM, Archer EW, Priebe N, Pillow JW (2013) Spectral methods for neural characterization using generalized quadratic models. In: Advances in neural information processing systems, pp 2454-2462.

Smola AJ, Schölkopf B (2004) A tutorial on support vector regression. Statistics and computing 14:199-222.

Tieleman T, Hinton G (2012) Lecture 6.5-RmsProp: Divide the gradient by a running average of its recent magnitude. COURSERA: Neural Networks for Machine Learning.

Wu W, Black MJ, Gao Y, Serruya M, Shaikhouni A, Donoghue J, Bienenstock E (2003) Neural decoding of cursor motion using a Kalman filter. In: Advances in neural information processing systems, pp 133-140.

Zhang K, Ginzburg I, McNaughton BL, Sejnowski TJ (1998) Interpreting neuronal population activity by reconstruction: unified framework with application to hippocampal place cells. J Neurophysiol 79:1017-1044.