

Machine learning for neural decoding

Joshua I. Glaser^{1,2,6,8,9*}, Ari S. Benjamin⁶, Raed H. Chowdhury^{3,4}, Matthew G. Perich^{3,4},
Lee E. Miller²⁻⁴, and Konrad P. Kording²⁻⁷

1. Interdepartmental Neuroscience Program, Northwestern University, Chicago, IL, USA
2. Department of Physical Medicine and Rehabilitation, Northwestern University and Shirley Ryan Ability Lab, Chicago, IL, USA
3. Department of Physiology, Northwestern University, Chicago, IL, USA
4. Department of Biomedical Engineering, Northwestern University, Chicago, IL, USA
5. Department of Applied Mathematics, Northwestern University, Chicago, IL, USA
6. Department of Bioengineering, University of Pennsylvania, Philadelphia, IL, USA
7. Department of Neuroscience, University of Pennsylvania, Philadelphia, IL, USA
8. Department of Statistics, Columbia University, New York, NY, USA
9. Zuckerman Mind Brain Behavior Institute, Columbia University, New York, NY, USA

* Contact: joshglaser88@gmail.com

Abstract:

Despite rapid advances in machine learning tools, the majority of neural decoding approaches still use traditional methods. Modern machine learning tools, which are versatile and easy to use, have the potential to significantly improve decoding performance. This tutorial describes how to effectively apply these algorithms for typical decoding problems. We provide descriptions, best practices, and code for applying common machine learning methods, including neural networks and gradient boosting. We also provide detailed comparisons of the performance of various methods at the task of decoding spiking activity in motor cortex, somatosensory cortex, and hippocampus. Modern methods, in particular neural networks and ensembles, significantly outperform traditional approaches, such as Wiener and Kalman filters. Improving the performance of neural decoding algorithms allows neuroscientists to better understand the information contained in a neural population, and can help advance engineering applications such as brain machine interfaces.

Introduction

Neural decoding uses activity recorded from the brain to make predictions about variables in the outside world. For example, researchers predict movements based on activity in motor cortex [1, 2], decisions based on activity in prefrontal and parietal cortices [3, 4], and spatial locations based on activity in the hippocampus [5, 6]. These decoding predictions can be used to control devices (e.g., a robotic limb), or to better understand how areas of the brain relate to the outside world. Decoding is a central tool in neural engineering and for neural data analysis.

In essence, neural decoding is a regression (or classification) problem relating neural signals to particular variables. When framing the problem in this way, it is apparent that there is a wide range of methods that one could apply. However, despite the recent advances in machine learning (ML) techniques for regression, it is still common to decode activity with traditional methods such as

linear regression. Using modern ML tools for neural decoding has the potential to boost performance significantly, and might allow deeper insights into neural function.

This tutorial is designed to help readers start applying standard ML methods for decoding. We describe when one should (or should not) use ML for decoding, how to choose an ML method, and best practices such as cross-validation and hyperparameter optimization. We provide companion code that makes it possible to implement a variety of decoding methods quickly. Using this same code, we demonstrate here that ML methods outperform traditional decoding methods. In example datasets of recordings from monkey motor cortex, monkey sensorimotor cortex, and rat hippocampus, modern ML methods showed the highest accuracy decoding of available methods. Using our code and this tutorial, readers can achieve these performance improvements on their own data.

Why use machine learning for decoding?

In general, modern machine learning has been shown to improve predictive performance greatly [7-9]. Based on this evidence of improved predictions (which we will demonstrate later in this tutorial in *A demonstration and comparison of machine learning for decoding*), there are multiple reasons to use ML to decode neural activity [10].

Engineering applications

Decoding is often used in engineering contexts, such as for brain machine interfaces (BMIs), where signals from motor cortex are used to control computer cursors [1], robotic arms [11], and muscles [2]. When the primary aim of these engineering applications is to improve predictive accuracy, ML should generally be beneficial¹.

Understanding what information is contained in neural activity

Decoding is also an important tool for understanding how neural signals relate to the outside world. It can be used to determine how much information neural activity contains about an external variable (e.g., sensation or movement) [12-14], and how this information differs across brain areas [15-17], experimental conditions [18, 19], and disease states [20]. When the goal is to determine how much information a neural population has about an external variable, regardless of the form of that information, then using ML will be beneficial.

While modern ML methods can provide an increase in decoding accuracy, it is important to be careful with the scientific interpretation of decoding results. Decoding can tell us how much information a neural population has about a variable X . However, high decoding accuracy does not mean that a brain area is directly involved in processing X , or that X is the purpose of the brain area. For example, with a powerful decoder, it could be possible to accurately classify images based on

¹ One problem that is often hypothesized for online applications, but whose magnitude is unclear, is that complicated machine learning algorithms could make it harder for users to learn to use their device.

recordings from the retina, since the retina has information about all visual space. However, this does not mean that the primary purpose of the retina is image classification. Moreover, even if the neural signal temporally precedes the external variable, it does not necessarily mean that it is causally involved. For example, movement-related information could reach somatosensory cortex prior to movement due to an efference copy from motor cortex, rather than somatosensory cortex being responsible for movement generation. Thus, researchers should constrain interpretations to merely address the information in neural populations about variables of interest, and how this information may vary across brain regions, experimental conditions, or time.

Benchmarking for simpler decoding models

Decoders can also be used to understand the form of the mapping between neural activity and variables in the outside world [21, 22]. That is, if researchers aim to test if the mapping from neural activity to behavior/stimuli (the “neural code”) has a certain structure, they can develop a “hypothesis-driven decoder” with a specific form. If that decoder can predict task variables with some arbitrary accuracy level, this is sometimes held as evidence that information within neural activity indeed has the hypothesized structure. However, it is important to know how well a hypothesis-driven decoder performs relative to what is possible. This is where modern ML methods can be of use. If a method designed to test a hypothesis decodes activity much worse than ML methods, then a researcher knows that their hypothesis likely misses key aspects of the neural code. Hypothesis-driven decoders should thus always be compared against a good-faith effort to maximize performance accuracy with a good machine learning approach.

What decoder should I use to improve predictive performance?

Depending on the recording method, location, and variables of interest, different decoding methods may be more effective. Neural networks, gradient boosted trees, support vector machines, and linear methods are among the dozens of potential candidates. Each makes different assumptions about how inputs relate to outputs. We will describe a number of specific methods suitable for neural decoding later in this tutorial. Ultimately, it is often beneficial to test multiple methods, perhaps starting with the methods we have found to work best for our demonstration datasets.

Different methods make different implicit assumptions about the data

The mapping between neural activity and task variables is likely to consistently have certain properties. For example, the mapping is likely smooth with respect to activity, meaning that tiny changes in population firing rates are associated with tiny changes in the task variables. As another example, task variables may change slowly in time. In general, when choosing a method with which to decode neural activity, it is important to choose a method that has assumptions that match the properties of the data. The stronger the assumptions that can be made, the better the decoder will perform — as long as those assumptions are correct.

There is no such thing as a method that makes no assumptions. This idea derives from a key theorem in the ML literature called the “No Free Lunch” theorem, which essentially states that no

algorithm will outperform all others on every problem [23]. The fact that some algorithms perform better than others in practice means that their assumptions about the data are better. Even the most complex neural networks make assumptions about how inputs relate to outputs, though these assumptions are weaker than the assumptions made by linear regression (in the sense that neural networks can also express nonlinear mappings). The knowledge that all methods make assumptions to varying degrees can help one be intentional about choosing a decoder.

For some neural decoding problems, the outputs are known to have some pattern or structure, and methods exist that allow this knowledge to be incorporated explicitly. For example, let us say we are estimating the kinematics of a movement from neural activity. The Kalman filter, which is frequently used for movement decoding [24-26], uses additional information about how movement kinematics transition over time. There are also Bayesian decoders that can incorporate prior beliefs about the decoded variables. For example, the Naive Bayes decoder, which is frequently used for position decoding from hippocampal activity [5, 27, 28], can take into account prior information about the probability distribution of positions. Decoders with task knowledge built in are constrained in terms of their solutions, which can be helpful, provided this knowledge is correct.

It is harder to know much about the structure of neural activity. How neural activity codes for the outside world (the “neural code”) is usually (if not always) unknown, and for this reason it is important to choose decoding algorithms that can express a wide array of input/output relationships. For example, if neural activity relates nonlinearly or nonmonotonically to the task variable, a method that can express nonlinear functions will outperform a linear method like a Kalman filter. In general, a good decoder will incorporate prior knowledge about the outputs, but still be nonlinear and relatively agnostic about the overall input/output relationship. It is nearly always better to make weaker assumptions about the data than to make strong assumptions that are wrong.

The perils of choosing a model that is too expressive or complex are visible in the phenomenon of “overfitting”, in which the model learns components of the noise that are unique to the data the model is trained on, but which do not generalize to any independent test data. To combat overfitting, one can choose a simpler algorithm that is less likely to learn to model this noise. A common intuition is that the number of free parameters in the algorithm (standing in for the method’s expressivity) should not exceed the number of training points. Alternatively, one can apply regularization, which essentially penalizes the complexity of a model. This effectively reduces the expressivity of a model, while still allowing it to have many free parameters [29]. Regularization is an important way to include the assumption that the input/output relationship is not arbitrarily complex.

Maximum likelihood estimates vs. posterior distributions

Different classes of methods also provide different types of estimates. Typically, machine learning algorithms provide maximum likelihood estimates of the decoded variable. That is, there is a single point estimate for the value of the decoded variable, which is the estimate that is most likely to be

true. Unlike the typical use of machine learning algorithms², Bayesian decoding provides a probability distribution over all possibilities for the decoded outputs (the “posterior” distribution), thus also providing information about the uncertainty of the estimate. The maximum likelihood estimate is the peak of the posterior distribution.

Ensemble methods

It is important to note that the user does not need to choose a single model. One can usually improve on the performance of any single model by combining multiple models, creating what is called an “ensemble”. Ensembles can be simple averages of the outputs of many models, or they can be more complex combinations. Weighted averages are common, which one can imagine as a second-tier linear model that takes the outputs of the first-tier models as inputs (sometimes referred to as a super learner). In principle, one can use any method as the second-tier model. Third- and fourth-tier models are uncommon but imaginable. Many successful ML methods are, at their core, ensembles [30]. In ML competition sites like Kaggle [31], most winning solutions are complicated ensembles.

Under which conditions will ML techniques improve decoding performance?

Whether ML will outperform other methods depends on many factors. The form of the underlying neural code, the length of the recording session, and the level of noise will all affect the predictive accuracy of different methods to different degrees. There is no way to know ahead of time which method will perform the best, and we recommend creating a pipeline to quickly test and compare many ML and simpler regression methods. For typical decoding situations, however, we expect certain ML methods to perform better than simpler methods. In the demonstration below, we show that this is true across three datasets and a wide range of variables like training data length, number of neurons, bin size, and hyperparameters. The reason is that, as a collection of methods, ML makes fewer assumptions about the form of the data. Many traditional methods assume that neural data relates linearly to the outputs, for example, or through a specific nonlinearity chosen beforehand. Machine learning, on the other hand, can learn how outputs relate to neural activity even if the relationship is quite nonlinear, and thus outperform other methods.

A practical guide for using machine learning for decoding

In any decoding problem, one has neural activity from multiple sources that is recorded for a period of time. While we focus here on spiking neurons, the same methods could be used with other forms of neural data, such as the BOLD signal in fMRI, or the power in particular frequency bands of local field potential (LFP) or electroencephalography (EEG) signals. When we decode from the neural data, whatever the source, we would like to predict the values of recorded outputs (Fig. 1a).

Data formatting/preprocessing for decoding

² While not commonly practiced, it is possible for modern ML algorithms to also estimate the uncertainty of the decoded variables. For example, a neural network can learn to estimate a variable along with its own uncertainty.

Preparing for regression vs. classification

Depending on the task, the desired output can be variables that are continuous (e.g., velocity or position), or discrete (e.g., choices). In the first case the decoder will perform regression, while in the second case it will perform classification. In our Python package, decoder classes are labeled to reflect this division.

In the data processing step, take note whether a prediction is desired continuously in time, or only at the end of each trial. In this tutorial, we focus on situations in which a prediction is desired continuously in time. However, many classification situations require only one prediction per trial (e.g., when making a single choice per trial). If this is the case, the data must be prepared such that many timepoints in a single trial are mapped to a single output.

Time binning divides continuous data into discrete chunks

A number of important decisions arise from the basic problem that time is continuously recorded, but decoding methods generally require discrete data for their inputs (and, in the continual-prediction situation, their outputs). A typical solution is to divide both the inputs and outputs into discrete time bins. These bins usually contain the average input or output over a small chunk of time, but could also contain the minimum, maximum, or the regular sampling of any interpolation method fit to the data.

When predictions are desired continuously in time, one needs to decide upon the temporal resolution, R , for decoding. That is, do we want to make a prediction every 50ms, 100ms, etc? We need to put the input and output into bins of length R (Fig. 1a). It is common (although not necessary) to use the same bin size for the neural data and output data, and we do so here. Thus, if T is the length of the recording, we will have approximately T/R total data points of neural activity and outputs.

Next, we need to choose the time period of neural activity used to predict a given output. In the simplest case, the activity from all neurons in a given time bin would be used to predict the output in that same time bin. However, it is often the case that we want the neural data to precede the output (e.g., in the case of making movements) or follow the decoder output (e.g., in the case of inferring the cause of a sensation). Plus, we often want to use neural data from more than one bin (e.g., using 500 ms of preceding neural data to predict a movement in the current 50 ms bin). In the following, we use the nomenclature that B time bins of neural activity are being used to predict a given output. For example, if we use one bin preceding the output, one concurrent bin, and one following bin, then $B=3$ (Fig. 1a). Note that when multiple bins of neural data are used to predict an output ($B>1$), then overlapping neural data will be used to predict different output times (Fig. 1a), making them dependent.

When multiple bins of neural data are used to predict an output, then we will need to exclude some output bins. For instance, if we are using one bin of neural data preceding the output, then we cannot predict the first output bin, and if we are using one bin of neural data following the output,

then we cannot predict the final output bin (Fig. 1a). Thus, we will be predicting K total output bins, where K is less than the total number of bins (T/R). To summarize, our decoders will be predicting each of these K outputs using B surrounding bins of activity from N neurons.

The format of the input data depends on the form of the decoder

Non-recurrent decoders: For most standard regression methods, the decoder has no persistent internal state or memory. In this case $N \times B$ features (the firing rates of each neuron in each relevant time bin) are used to predict each output (Fig. 1b). The input matrix of covariates, \mathbf{X} , has $N \times B$ columns (one for each feature) and K rows (corresponding to each output being predicted). If there is a single output that is being predicted, it can be put in a vector, \mathbf{Y} , of length K . Note that for many decoders, if there are multiple outputs, each is independently decoded. If multiple outputs are being simultaneously predicted, which can occur with neural network decoders, the outputs can be put in a matrix \mathbf{Y} , that has K rows and d columns, where d is the number of outputs being predicted. Since this is the format of a standard regression problem, many regression methods can easily be substituted for one another once the data has been prepared in this way.

Recurrent neural network decoders: When using recurrent neural networks (RNNs) for decoding, we need to put the inputs in a different format. Recurrent neural networks explicitly model temporal transitions across time with a persistent internal state, called the “hidden state” (Fig. 1c). At each time of the B time bins, the hidden state is adjusted as a function of both the N features (the firing rates of all neurons in that time bin) and the hidden state at the previous time bin (Fig. 1c). After transitioning through all B bins, the hidden state in this final bin is used to predict the output. In this way an RNN decoder can integrate the effect of neural inputs over an extended period of time. For use in this type of decoder, the input can be formatted as a 3-dimensional tensor of size $K \times N \times B$ (Fig. 1c). That is, for each row (corresponding to one of the K output bins to be predicted), there will be N features (2nd tensor dimension) over B bins (3rd tensor dimension) used for prediction. Within this format, different types of RNNs, including those more sophisticated than the standard RNN shown in Fig. 1c, can be easily switched for one another.

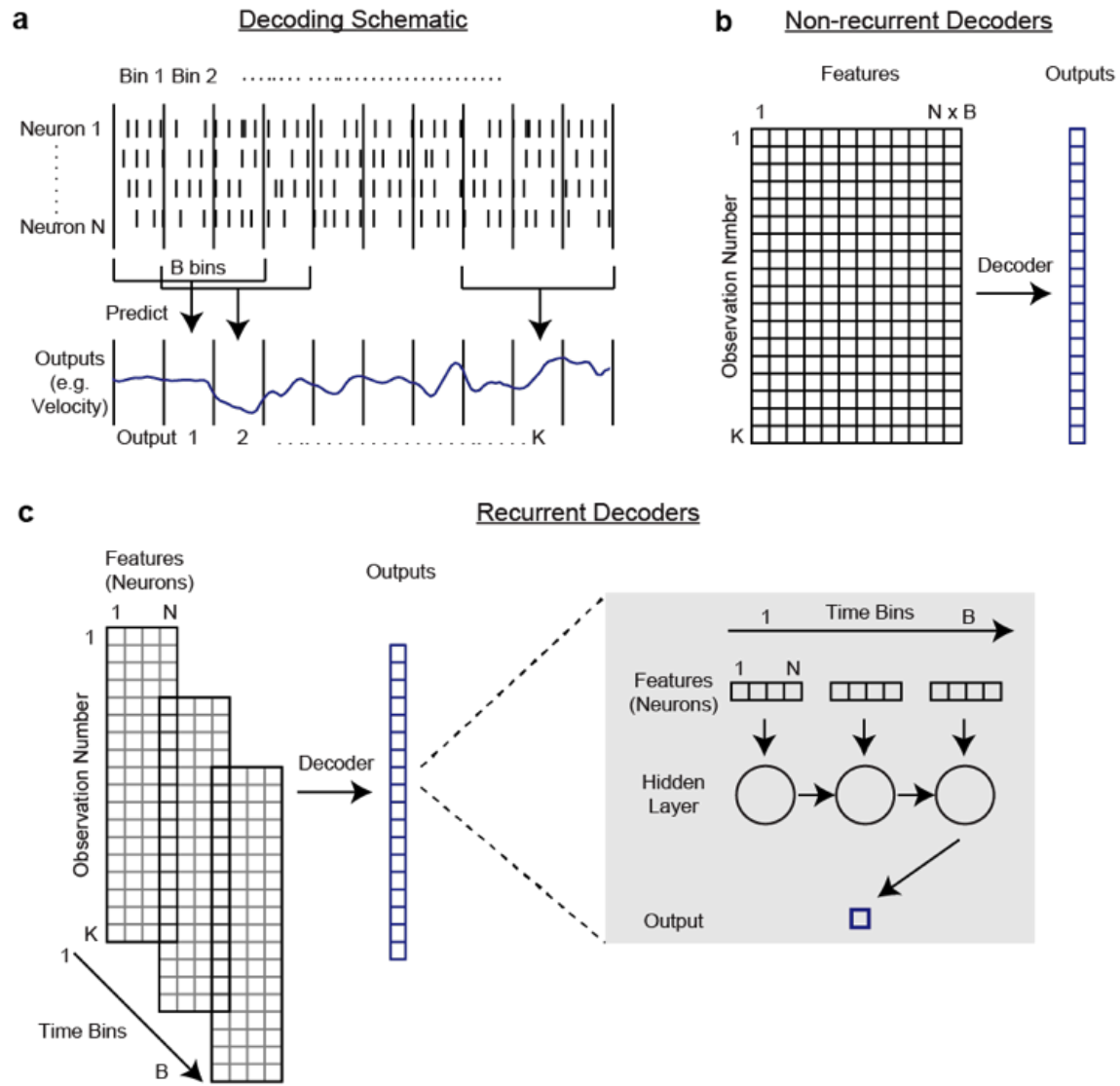


Figure 1: Decoding Schematic

a) To decode (predict) the output in a given time bin, we used the firing rates of all N neurons in B time bins. In this schematic, $N=4$ and $B=3$ (one bin preceding the output, one concurrent bin, and one following bin). Here, we show a single output being predicted. **b)** For non-recurrent decoders (Wiener Filter, Wiener Cascade, Support Vector Regression, XGBoost, and Feedforward Neural Network in our subsequent demonstration), this is a standard machine learning regression problem where $N \times B$ features (the firing rates of each neuron in each relevant time bin) are used to predict the output. **c)** To predict outputs with recurrent decoders (simple recurrent neural network, GRUs, LSTMs in our subsequent demonstration) we used N features, with temporal connections across B bins. A schematic of a recurrent neural network predicting a single output is on the right.

Applying machine learning

The typical process of applying ML to data involves testing several methods and seeing which works best. Some methods may be expected to work better or worse depending on the structure of

the data, and in the next part of this tutorial we provide a demonstration of which methods work best for typical neural spiking datasets. That said, applying ML is unavoidably an iterative process, and for this reason we begin with the proper way to choose among many methods. It is particularly important to consider how to avoid overfitting our data during this iterative process.

Compare method performance using cross-validation

Given two (or more) methods, the ML practitioner must have a principled way to decide which method is best. It is crucial to test the decoder performance on a separate, held-out dataset that the decoder did not see during training (Fig. 2a). This is because a decoder typically will *overfit* to its training data. That is, it will learn to predict outputs using idiosyncratic information about each datapoint in the training set, such as noise, rather than the aspects that are general to the data. An overfit algorithm will describe the training data well but will not be able to provide good predictions on new datasets. For this reason, the proper metric to choose between methods is the performance on held-out data, meaning that the available data should be split into separate “training” and “testing” datasets. In practice, this will reduce the amount of training data available. In order to efficiently use all of the data, it is common to perform cross-validation (Fig. 2b). In 10-fold cross-validation, for example, the dataset is split into 10 sets. The decoder is trained on 9 of the sets, and performance is tested on the final set. This is done 10 times in a rotating fashion, so that each set is tested once. The performance on all test sets is generally averaged together to determine the overall performance.

When publishing about the performance of a method, it is important to show the performance on held-out data that was additionally not used to select between methods (Fig. 2c). Random noise may lead some methods to perform better than others, and it is possible to fool oneself and others by testing a great number of methods and choosing the best. In fact, one can entirely overfit to a testing dataset simply by using the test performance to select the best method, even if no method is trained on that data explicitly. Practitioners often make the distinction between the “training” data, the “validation” data (which is used to select between methods) and the “testing” data (which is used to evaluate the true performance). This three-way split of data complicates cross-validation somewhat (Fig. 2b). It is possible to first create a separate testing set, then choose among methods on the remaining data using cross-validation. Alternatively, for maximum data efficiency, one can iteratively rotate which split is the testing set, and thus perform a two-tier nested cross-validation.

Hyperparameter optimization

When fitting any single method, one usually has to additionally choose a set of *hyperparameters*. These are parameters that relate to the design of the decoder itself, and should not be confused with the ‘normal’ parameters that are fit during optimization, e.g., the weights that are fit in linear regression. For example, neural networks can be designed to have any number of hidden units. Thus, the user needs to set the number of hidden units (the hyperparameter) before training the decoder. Often decoders have multiple hyperparameters, and different hyperparameter values can sometimes lead to greatly different performance. Thus, it is important to choose a decoder’s hyperparameters carefully.

When using a decoder that has hyperparameters, one should take the following steps. First, always split the data into three separate sets (training set, testing set, and validation set), perhaps using nested cross-validation (Fig. 2b). Next, iterate through a large number of hyperparameter settings and choose the best based on validation set performance. Simple methods for searching through hyperparameters are a grid search (i.e., sampling values evenly) and random search [32]. There are also more efficient methods (e.g., [33, 34]) that can intelligently search through hyperparameters based on the performance of previously tested hyperparameters. The best performing hyperparameter and method combination (on the validation set) will be the final method, unless one is combining multiple methods into an ensemble.

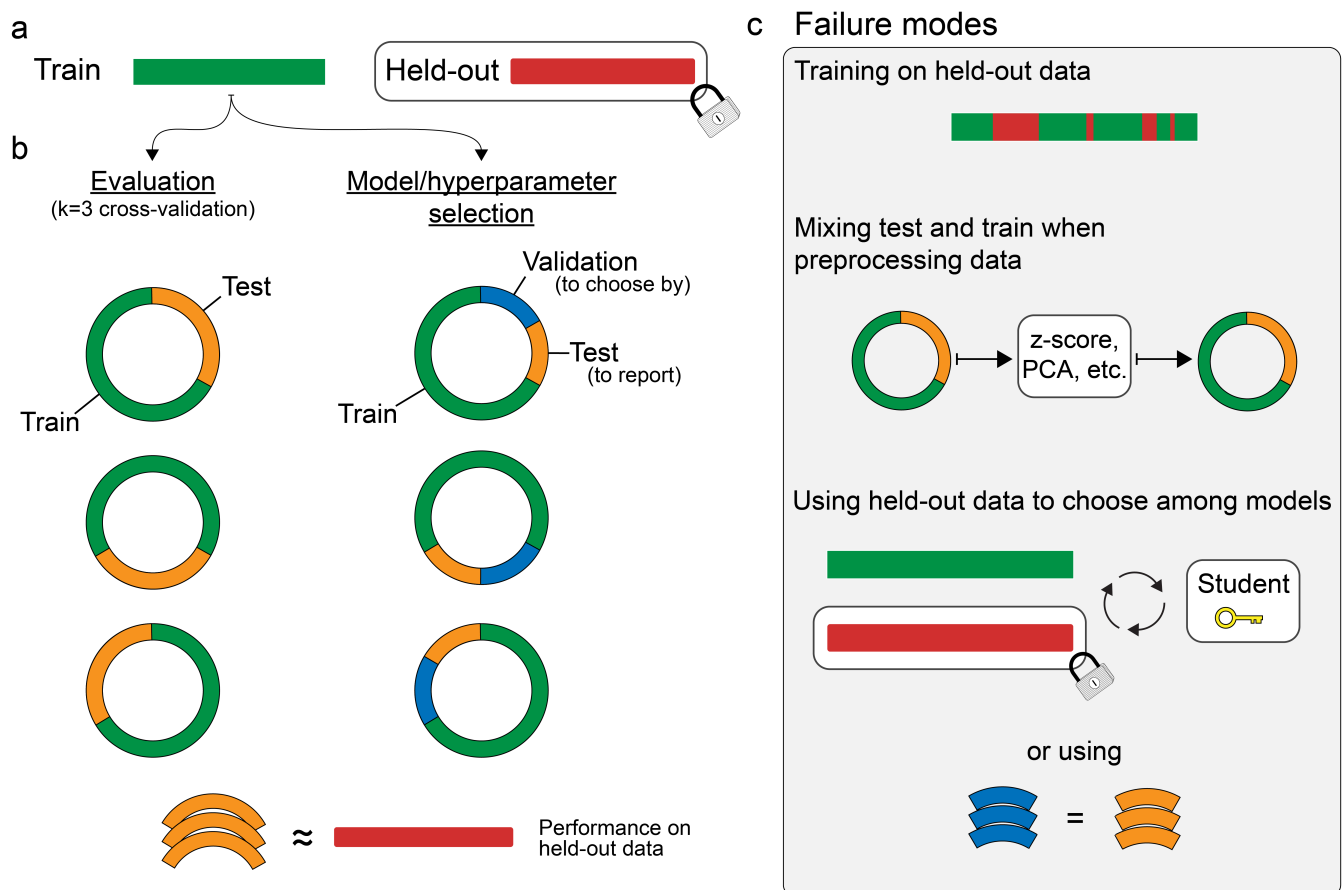


Figure 2. Schematic of cross-validation

a) After training a decoder on some data (green), we would like to know how well it performs on held-out data that we do not have access to (red). **b) Left:** By splitting the data we have into test (orange) and train (green) segments, we can approximate the performance on held-out data. In k-fold cross-validation, we re-train the decoder k times, and each time rotate which parts of the data are the test or train data. The average test set performance approximates the performance on held-out data. **Right:** If we want to select among many models, we cannot maximize the performance on the same data we will report as a score. (This is very similar to “p-hacking” statistical significance.) Instead we maximize performance on “validation” data (blue),

and again rotate through the available data. **c)** All failure modes are ways in which a researcher lets information from the test set “leak” into the training algorithm. This happens if you explicitly train on the test data (top), or use any statistics of the test data to modify the train data before fitting (middle), or select your models or hyperparameters based on the performance on the test data (bottom).

A demonstration and comparison of machine learning for decoding

We have prepared a Python package that implements the machine learning pipeline for decoding problems. It is available at https://github.com/KordingLab/Neural_Decoding, and includes code to correctly format the neural and output data for decoding, to implement many different decoders for both regression and classification, and to optimize their hyperparameters. In this section, we demonstrate its usage, and show which of its methods work better than traditional methods for several neural datasets.

Note that additional details, discussion, and demonstrations can be found in the Supplemental Materials.

Specific decoders:

The following decoders are available in our package, and we demonstrate their performance on example datasets below. We included both historical linear techniques (e.g., the Wiener filter) and modern ML techniques (e.g., neural networks and ensembles of techniques). More details about the precise implementation of these methods can be found in Supplemental Information.

Wiener Filter: The Wiener filter uses multiple linear regression to predict the output from multiple time bins of all neurons’ spikes. That is, the output is assumed to be a linear mapping of the number of spikes in the relevant time bins from every neuron (Fig. 1a,b).

Wiener Cascade: The Wiener cascade (also known as a linear-nonlinear model) fits a linear regression (the Wiener filter) followed by a fitted static nonlinearity (e.g., [35]). This allows for a nonlinear relationship between the input and the output, and assumes that this nonlinearity is purely a function of the linear output. The default nonlinear component is a polynomial with degree that can be determined on a validation set.

Support Vector Regression: In support vector regression (SVR) [36], the inputs are projected into a higher-dimensional space using a nonlinear kernel, and then linearly mapped from this space to the output to minimize an objective function [36]. In our toolbox, the default kernel is a radial basis function.

XGBoost: XGBoost (Extreme Gradient Boosting) [37] is an implementation of gradient boosted trees. Tree-based methods sequentially split the input space into many discrete parts (visualized as branches on a tree for each split), in order to assign each final “leaf” (a portion of input space that is not split any more) a value in output space [38]. XGBoost fits many regression trees, which are trees that predict continuous output values. “Gradient boosting” refers to fitting each subsequent regression tree to the residuals of the previous fit.

Feedforward Neural Network: A feedforward neural net connects the inputs to sequential layers of hidden units, which then connect to the output. Each layer connects to the next (e.g., the input layer to the first hidden layer, or the first to second hidden layers) via linear mappings followed by nonlinearities. Note that the Wiener cascade is a special case of a neural network with no hidden layers.

Simple RNN: In a standard recurrent neural network (RNN), the hidden state is a linear combination of the inputs and the previous hidden state. This hidden state is then run through an output nonlinearity, and linearly mapped to the output. RNNs, unlike feedforward neural networks, allow temporal changes in the system to be modeled explicitly.

Gated Recurrent Unit: Gated recurrent units (GRUs) [39] are a more complex type of recurrent neural network. It has gated units, which determine how much information can flow through various parts of the network. In practice, these gated units allow for better learning of long-term dependencies.

Long Short Term Memory Network: Like the GRU, the long short term memory (LSTM) network [40] is a more complex recurrent neural network with gated units that further improve the capture of long-term dependencies. The LSTM has more parameters than the GRU.

Kalman Filter: Our Kalman filter for neural decoding was based on [24]. In the Kalman filter, the hidden state at time t is a linear function of the hidden state at time $t-1$, plus a matrix characterizing the uncertainty. For neural decoding, the hidden state is the kinematics (x and y components of position, velocity, and acceleration). Note that even though we only aim to predict position or velocity, all kinematics are included because this allows for better prediction.

Naïve Bayes: The Naïve Bayes decoder is a type of Bayesian decoder that determines the probabilities of different outcomes, and it then predicts the most probable. Briefly, it fits an encoding model to each neuron, makes conditional independence assumptions about neurons, and then uses Bayes' rule to create a decoding model from the encoding models [5]. This probabilistic framework can incorporate prior information about the variables.

In the comparisons below, we also demonstrate an ensemble method. We combined the predictions from all decoders except the Kalman filter and Naïve Bayes decoders (which have different formats) using a feedforward neural network. That is, the eight methods' predictions were provided as input into a feedforward neural network that we trained to predict the true output.

Demonstration datasets

We first examined which of several decoding methods performed the best across three datasets from motor cortex, somatosensory cortex, and hippocampus.

In the task for decoding from motor cortex, monkeys moved a manipulandum that controlled a cursor on a screen [19], and we aimed to decode the x and y velocity of the cursor (Fig. S1). The 21 minute recording from motor cortex contained 164 neurons. The mean and median firing rates, respectively, were 6.7 and 3.4 spikes / sec. Data were put into 50 ms bins. We used 700 ms of neural activity (13 bins before and the concurrent bin) to predict the current movement velocity.

The same task was used in the recording from somatosensory cortex [41]. The recording from S1 was 51 minutes, and contained 52 neurons. The mean and median firing rates, respectively, were 9.3 and 6.3 spikes / sec. Data were put into 50 ms bins. We used 650 ms surrounding the movement (6 bins before, the concurrent bin, and 6 bins after).

In the task for decoding from hippocampus, rats chased rewards on a platform [42, 43], and we aimed to decode the rat's x and y position (Fig. S1). From this recording, we used 46 neurons over a time period of 75 minutes. These neurons had mean and median firing rates of 1.7 and 0.2 spikes / sec, respectively. Data were put into 200 ms bins. We used 2 seconds of surrounding neural activity (4 bins before, the concurrent bin, and 5 bins after) to predict the current position. Note that the bins used for decoding differed in all tasks for the Kalman filter and Naïve Bayes decoders (see *Supplemental Information* for additional details).

Performance comparison

In order to get a qualitative impression of the performance, we first plotted the output of each decoding method for each of the three datasets (Fig. 3). In these examples, the modern methods, such as the LSTM and ensemble, appeared to outperform traditional methods. We next quantitatively compared the methods' performances, using the metric of R^2 on held-out test sets (see *Supplemental Information* for details). These results confirmed our qualitative findings (Fig. 4). In particular, neural networks and the ensemble led to the best performance, while the Wiener or Kalman Filter led to the worst performance. In fact, the LSTM decoder explained over 40% of the unexplained variance from a Wiener filter (R^2 's of 0.88, 0.86, 0.62 vs. 0.78, 0.75, 0.35). Interestingly, while the Naïve Bayes decoder performed relatively well when predicting position in the hippocampus dataset (mean R^2 just slightly less than the neural networks), it performed very poorly when predicting hand velocities in the other two datasets. Another interesting finding is that the feedforward neural network did almost as well as the LSTM in all brain areas. Across cases, the ensemble method added a reliable, but small increase to the explained variance. Overall, modern ML methods led to significant increases in predictive power.

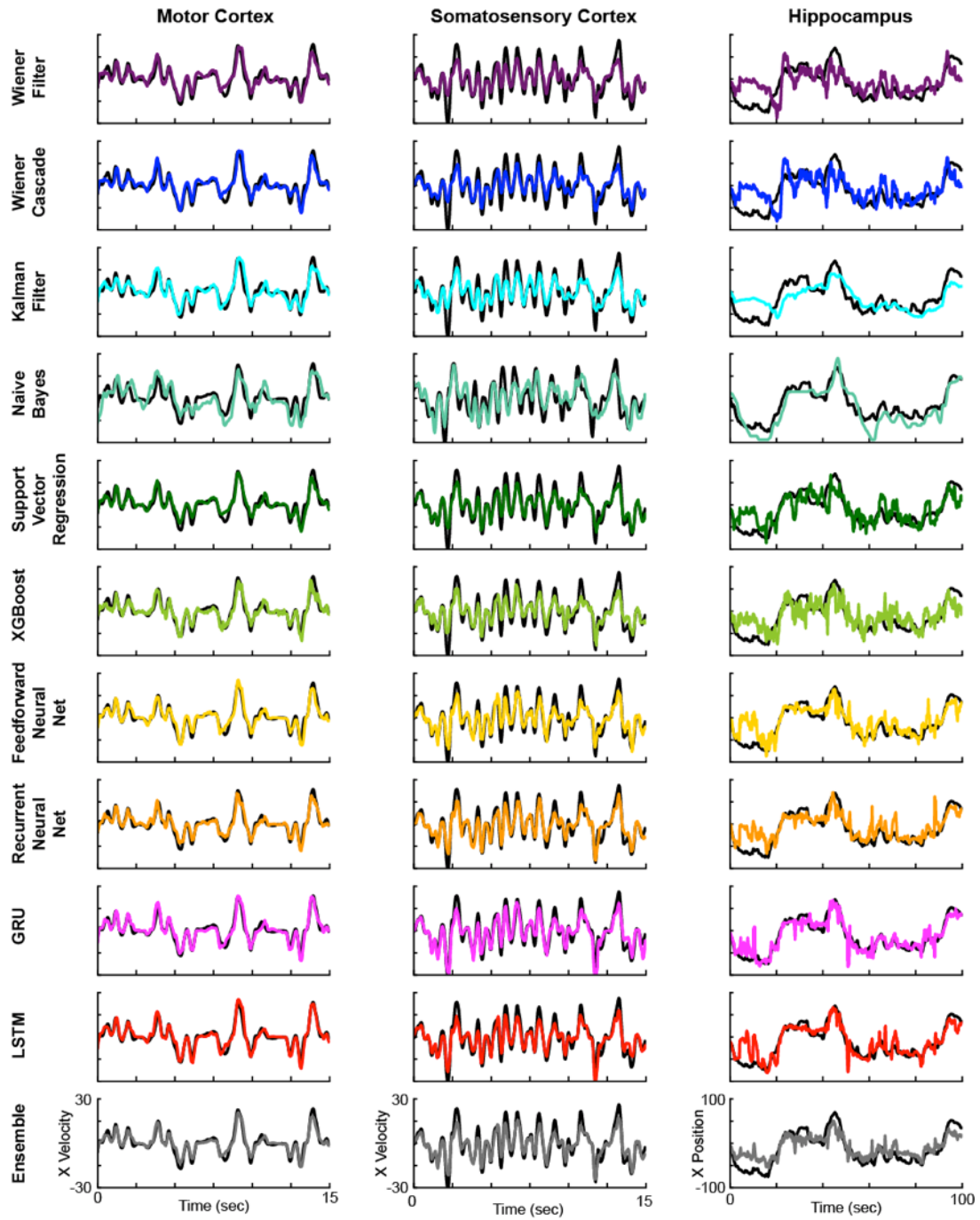


Figure 3: Example Decoder Results

Example decoding results from motor cortex (left), somatosensory cortex (middle), and hippocampus (right), for all eleven methods (top to bottom). Ground truth traces are in black, while decoder results are in various colors.

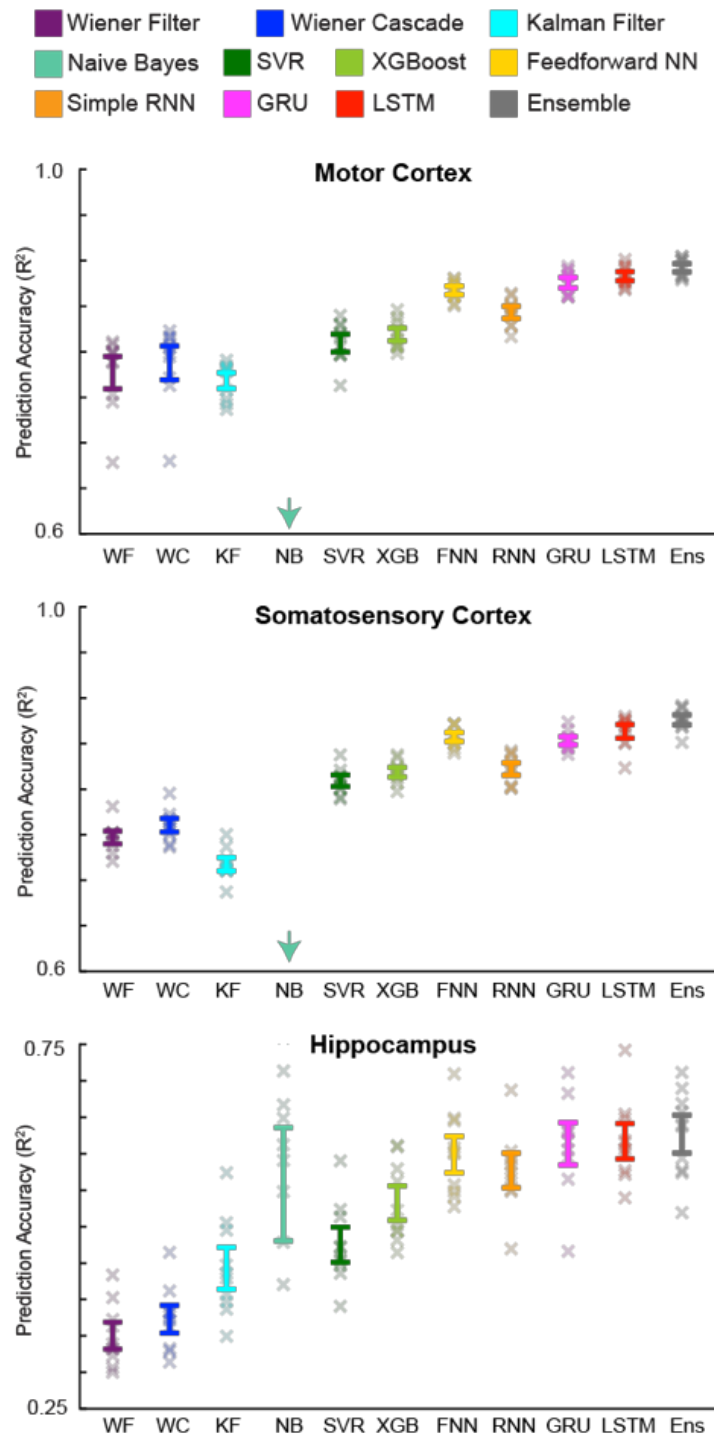


Figure 4: Decoder Result Summary

R^2 values are reported for all decoders (different colors) for each brain area (top to bottom). Error bars represent the mean \pm SEM across cross-validation folds. X's represent the R^2 values of each cross-validation fold. The NB decoder had mean R^2 values of 0.26 and 0.36 (below the minimum y-axis value) for the motor and somatosensory cortex datasets, respectively. Note the different y-axis limits for the hippocampus dataset in this and all subsequent figures.

Concerns about limited data for decoding

We chose a representative subset of the ten methods to pursue further questions about particular aspects of neural data analysis: the feedforward neural network and LSTM (two effective modern methods), along with the Wiener and Kalman filters (two traditional methods in widespread use). The improved predictive performance of the modern methods is likely due to their greater complexity. However, this greater complexity may make these methods unsuitable for smaller amounts of data. Thus, we tested performance with varying amounts of training data. With only 2 minutes of data for motor and somatosensory cortices, and 15 minutes of hippocampus data, both modern methods outperformed both traditional methods (Fig. 5a, Fig. S2). When decreasing the amount of training data further, to only 1 minute for motor and somatosensory cortices and 7.5 minutes for hippocampus data, the Kalman filter performance was sometimes comparable to the modern methods, but the modern methods significantly outperformed the Wiener Filter (Fig. 5a). Thus, even for limited recording times, modern ML methods can yield significant gains in decoding performance.

Besides limited recording times, neural data is often limited in the number of recorded neurons. Thus, we compared methods using a subset of only 10 neurons. For motor and somatosensory data, despite a general decrease in performance for all decoding methods, the modern methods significantly outperformed the traditional methods (Fig. 5b). For the hippocampus dataset, no method predicted well (mean $R^2 < 0.25$) with only 10 neurons. This is likely because 10 sparsely firing neurons (median firing of HC neurons was ~ 0.2 spikes / sec) did not contain enough information about the entire space of positions. However, in most scenarios, with limited neurons and for limited recorded times, modern ML methods can be advantageous.

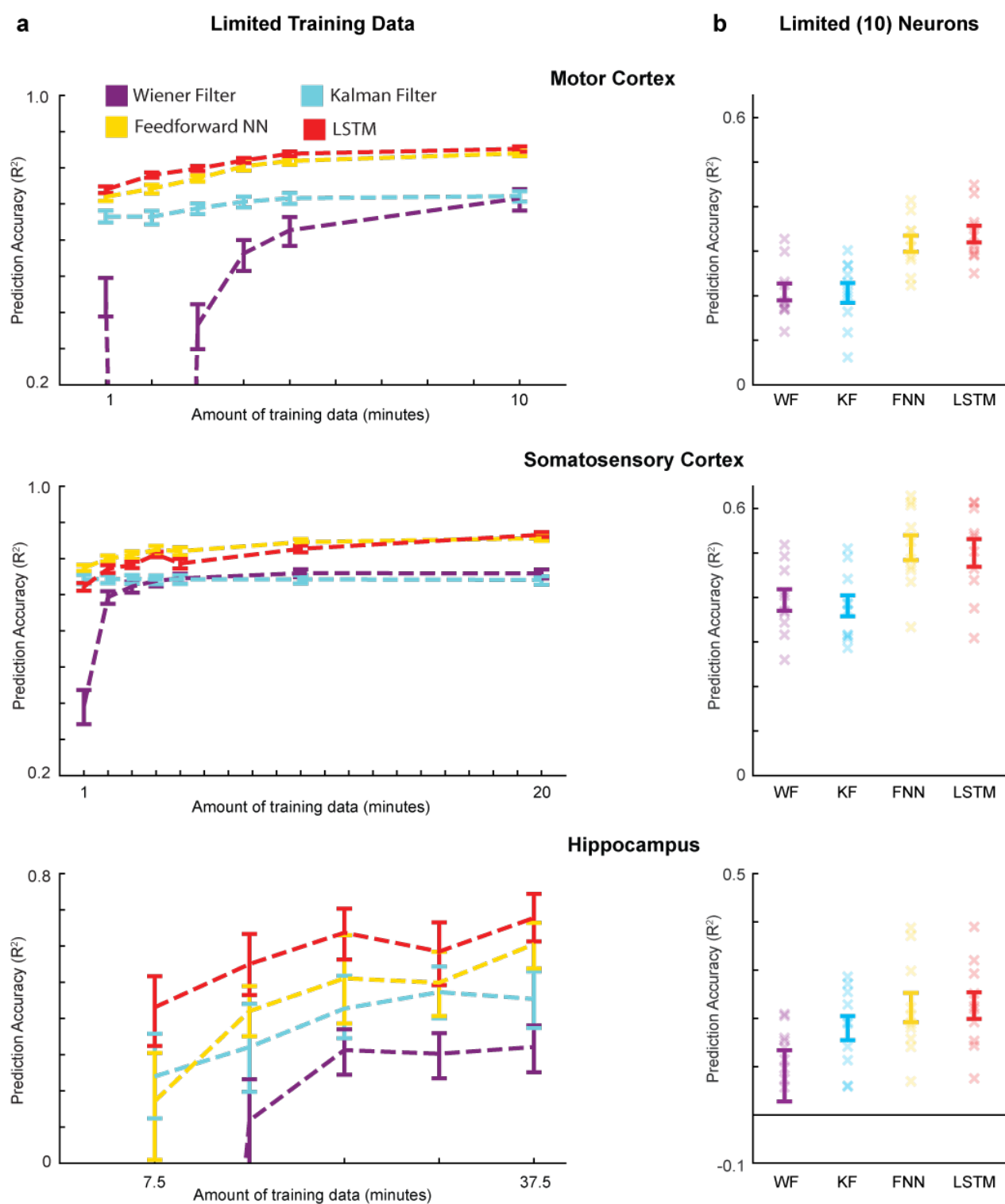


Figure 5: Decoder results with limited data

(A) Testing the effects of limited training data. Using varying amounts of training data, we trained two traditional methods (Wiener filter and Kalman filter), and two modern methods (feedforward neural network and LSTM). R^2 values are reported for these decoders (different colors) for each brain area (top to bottom). Error bars are 68% confidence intervals (meant to approximate the SEM) produced via bootstrapping, as we used a single test set. Values with negative R^2 s were not shown. **(B)** Testing the effects

of few neurons. Using only 10 neurons, we trained two traditional methods (Wiener filter and Kalman filter), and two modern methods (feedforward neural network and LSTM). We used the same testing set as in panel A, and the largest training set from panel A. R^2 values are reported for these decoders for each brain area. Error bars represent the mean \pm SEM of multiple repetitions with different subsets of 10 neurons. X's represent the R^2 values of each repetition. Note that the y-axis limits are different in panels A and B.

Concerns about run-time

While modern ML methods lead to improved performance, it is important to know that these sophisticated decoding models can be trained in a reasonable amount of time. To get a feeling for the typical timescale, consider that when running our demonstration data on a desktop computer using only CPUs, it took less than 1 second to fit a Wiener filter, less than 10 seconds to fit a feedforward neural, and less than 8 minutes to fit an LSTM. (This was for 30 minutes of data, using 10 time bins of neural activity for each prediction, and 50 neurons.) In practice, these models will need to be fit tens to hundreds of times when incorporating hyperparameter optimization and cross-validation. As a concrete example, let us say that we are doing 5-fold cross-validation, and hyperparameter optimization requires 50 iterations per cross-validation fold. In that case, fitting the LSTM would take about 30 hours, and fitting the feedforward neural net would take less than 1 hour. Note that variations in hardware and software implementations can drastically change runtime, and modern GPUs can often increase speeds approximately 10-fold. Thus, while modern methods do take significantly longer to train (and may require running overnight without GPUs), that should be manageable for most offline applications.

Concerns about robustness to hyperparameters

All our previous results used hyperparameter optimization. While we strongly encourage a thorough hyperparameter optimization, a user with limited time might just do a limited hyperparameter search. Thus, it is helpful to know how sensitive results may be to varying hyperparameters. We tested the performance of the feedforward neural network while varying two hyperparameters: the number of units and the dropout rate (a regularization hyperparameter for neural networks). We held the third hyperparameter in our code package, the number of training epochs, constant at 10. We found that the performance of the neural network was generally robust to large changes in the hyperparameters (Fig. 6). As an example, for the somatosensory cortex dataset, the peak performance of the neural network was $R^2=0.86$ with 1000 units and 0 dropout, and virtually the same ($R^2=0.84$) with 300 units and 30% dropout. Even when using limited data, neural network performance was robust to hyperparameter changes. For instance, when training the somatosensory cortex dataset with 1 minute of training data, the peak performance was $R^2=0.77$ with 700 units and 20% dropout. A network with 300 units and 30% dropout had $R^2=0.75$. Note that the hippocampus dataset, in particular when using limited training data, did have greater variability, emphasizing the importance of hyperparameter optimization on sparse datasets. However, for most datasets, researchers should not be concerned that slightly non-optimal hyperparameters will lead to largely degraded performance.

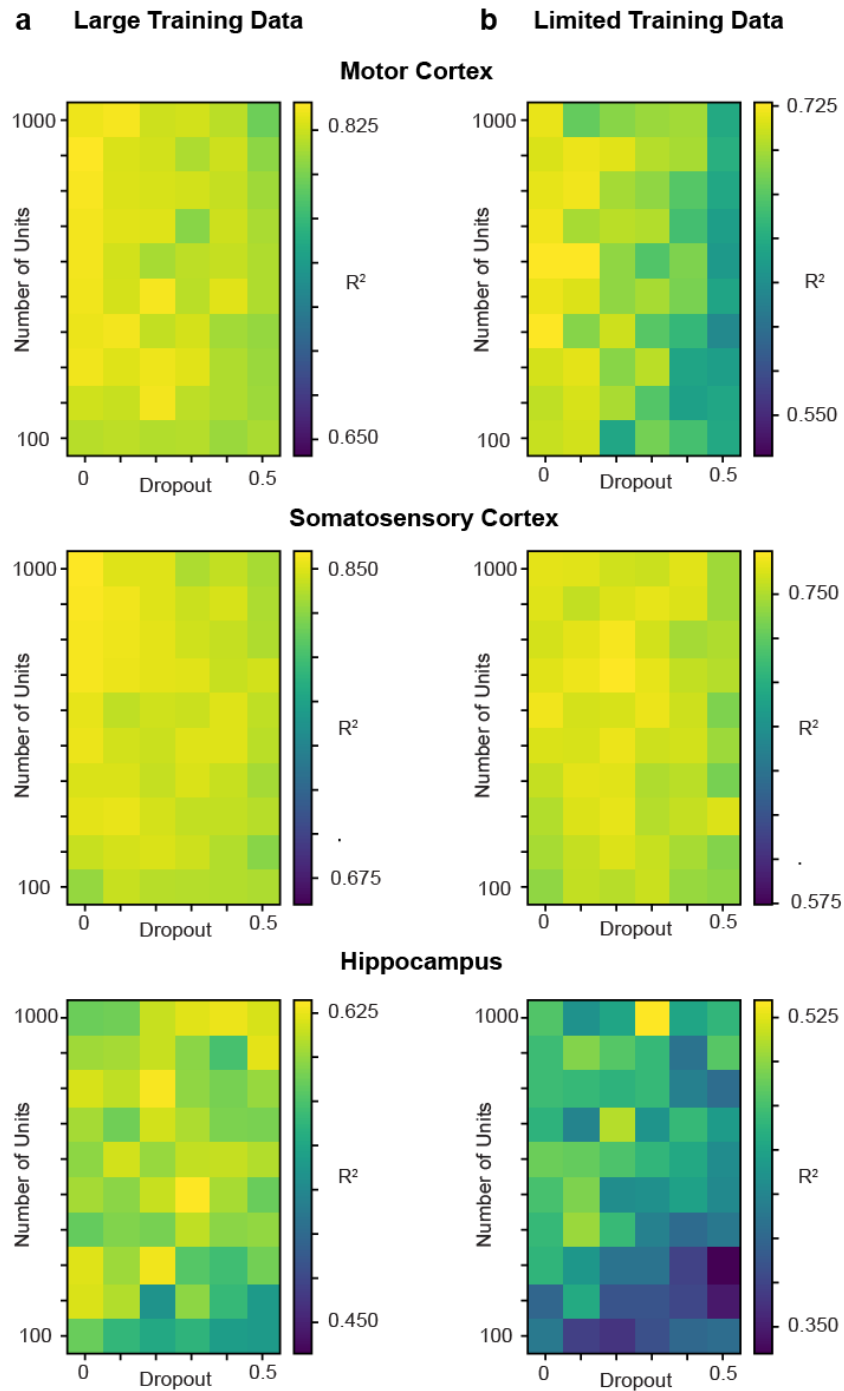


Figure 6: Sensitivity of neural network results to hyperparameter selection

In a feedforward neural network, we varied the number of hidden units per layer (in increments of 100) and the proportion of dropout (in increments of 0.1), and evaluated the decoder's performance on all three datasets (top to bottom). The neural network had two hidden layers, each with the same number of hidden units. The number of training epochs was kept constant at 10. The colors show the R^2 on the test set, and each panel's colors were put in the range: $[\text{maximum } R^2 - 0.2, \text{maximum } R^2]$. **a)** We used a large amount of training data (the maximum amount used in Fig. 5a), which was 10, 20, and 37.5 minutes of data for the motor cortex, somatosensory cortex, and hippocampus datasets, respectively. **b)** Same results for a limited amount of training data: 1, 1, and 15 minutes of data for the motor cortex, somatosensory cortex, and hippocampus datasets, respectively.

Discussion of this demonstration

Our comparisons, which were made using the code we have published online, show that machine learning works well on typical neural decoding datasets, outperforming traditional decoding methods. In our demonstration, we decoded continuous-valued variables. However, these same methods can be used for classification tasks, which often use classic decoders such as logistic regression and support vector machines. Our available code also includes classification methods.

We have decoded from spiking data, but it is possible that the problem of decoding from other data modalities is different. One main driver of a difference may be the distinct levels of noise. For example, fMRI signals have far higher noise levels than spikes. As the noise level goes up, linear techniques become more appropriate, which may ultimately lead to a situation where the traditional linear techniques become superior. Applying the same analyses we did here across different data modalities is an important next step.

All our decoding was done “offline,” meaning that the decoding occurred after the recording, and was not part of a control loop. This type of decoding is useful for determining how information in a particular brain area relates to an external variable. However, for engineering applications such as BMIs [44, 45], the goal is to decode information (e.g., predict movements) in real time. Our results here may not apply as directly to online decoding situations, since the subject is ultimately able to adapt to imperfections in the decoder. In that case, even relatively large decoder performance differences may be irrelevant. Plus, there are additional challenges in online applications, such as non-stationary inputs (e.g., due to electrodes shifting in the brain) [25, 46, 47]. Finally, online applications are concerned with computational runtime, which we have only briefly addressed here. In the future, it would be valuable to test modern techniques for decoding in online applications (as in [46, 48]).

Conclusion

Machine learning is relatively straightforward to apply and can greatly improve the performance of neural decoding. Its principle advantage is that many fewer assumptions need to be made about the structure of the neural activity and the decoded variables. Best practices like testing on held-out data, perhaps via crossvalidation, are crucial to the ML pipeline and are critical for any application. The Python package that accompanies this tutorial is designed to guide both best practices and the deployment of specific ML algorithms, and we expect it will be useful in improving decoding performance for new datasets. Our hunch is that it will be hard for specialized algorithms (e.g. [49, 50]) to compete with the standard algorithms developed by the machine learning community.

Acknowledgements

We would like to thank Pavan Ramkumar for help with code development. For funding, JG was supported by NIH F31 EY025532 and NIH T32 HD057845. AB was supported by NIH MH103910.

MP was supported by NIH F31 NS092356 and NIH T32 HD07418. RC was supported by NIH R01 NS095251 and DGE-1324585. LM was supported by NIH R01 NS074044 and NIH R01 NS095251. KK was supported by NIH R01 NS074044, NIH R01 NS063399 and NIH R01 EY021579.

References

1. Serruya MD, Hatsopoulos NG, Paninski L, Fellows MR, Donoghue JP. Brain-machine interface: Instant neural control of a movement signal. *Nature*. 2002;416(6877):141-2.
2. Ethier C, Oby ER, Bauman MJ, Miller LE. Restoration of grasp following paralysis through brain-controlled stimulation of muscles. *Nature*. 2012;485(7398):368-71. doi: 10.1038/nature10987. PubMed PMID: 22522928; PubMed Central PMCID: PMC3358575.
3. Baeg E, Kim Y, Huh K, Mook-Jung I, Kim H, Jung M. Dynamics of population code for working memory in the prefrontal cortex. *Neuron*. 2003;40(1):177-88.
4. Ibos G, Freedman DJ. Sequential sensory and decision processing in posterior parietal cortex. *eLife*. 2017;6.
5. Zhang K, Ginzburg I, McNaughton BL, Sejnowski TJ. Interpreting neuronal population activity by reconstruction: unified framework with application to hippocampal place cells. *J Neurophysiol*. 1998;79(2):1017-44.
6. Davidson TJ, Kloosterman F, Wilson MA. Hippocampal replay of extended experience. *Neuron*. 2009;63(4):497-507.
7. He K, Zhang X, Ren S, Sun J. Delving Deep into Rectifiers: Surpassing Human-Level Performance on ImageNet Classification. 2015 IEEE International Conference on Computer Vision (ICCV); 20152015.
8. Silver D, Huang A, Maddison CJ, Guez A, Sifre L, van den Driessche G, et al. Mastering the game of Go with deep neural networks and tree search. *Nature*. 2016;529(7587):484-9. doi: 10.1038/nature16961.
9. LeCun Y, Bengio Y, Hinton G. Deep learning. *Nature*. 2015;521(7553):436-44. doi: 10.1038/nature14539. PubMed PMID: 26017442.
10. Glaser JI, Benjamin AS, Farhoodi R, Kording KP. The roles of supervised machine learning in systems neuroscience. *Prog Neurobiol*. 2019.
11. Collinger JL, Wodlinger B, Downey JE, Wang W, Tyler-Kabara EC, Weber DJ, et al. High-performance neuroprosthetic control by an individual with tetraplegia. *The Lancet*. 2013;381(9866):557-64.
12. Raposo D, Kaufman MT, Churchland AK. A category-free neural population supports evolving demands during decision-making. *Nat Neurosci*. 2014;17(12):1784-92.
13. Rich EL, Wallis JD. Decoding subjective decisions from orbitofrontal cortex. *Nat Neurosci*. 2016;19(7):973-80.
14. Hung CP, Kreiman G, Poggio T, DiCarlo JJ. Fast readout of object identity from macaque inferior temporal cortex. *Science*. 2005;310(5749):863-6.
15. Quiroga RQ, Snyder LH, Batista AP, Cui H, Andersen RA. Movement intention is better predicted than attention in the posterior parietal cortex. *J Neurosci*. 2006;26(13):3615-20.
16. Hernández A, Nácher V, Luna R, Zainos A, Lemus L, Alvarez M, et al. Decoding a perceptual decision process across cortex. *Neuron*. 2010;66(2):300-14.
17. van der Meer MA, Johnson A, Schmitzer-Torbert NC, Redish AD. Triple dissociation of information processing in dorsal striatum, ventral striatum, and hippocampus on a learned spatial decision task. *Neuron*. 2010;67(1):25-32.
18. Dekleva BM, Ramkumar P, Wanda PA, Kording KP, Miller LE. Uncertainty leads to persistent effects on reach representations in dorsal premotor cortex. *eLife*. 2016;5:e14316. doi: 10.7554/eLife.14316.
19. Glaser JI, Perich MG, Ramkumar P, Miller LE, Kording KP. Population coding of conditional probability distributions in dorsal premotor cortex. *Nature Communications*. 2018;9(1):1788. doi: 10.1038/s41467-018-04062-6.
20. Weygandt M, Blecker CR, Schäfer A, Hackmack K, Haynes J-D, Vaitl D, et al. fMRI pattern recognition in obsessive-compulsive disorder. *Neuroimage*. 2012;60(2):1186-93.
21. Naufel S, Glaser JI, Kording KP, Perreault EJ, Miller LE. A muscle-activity-dependent gain between motor cortex and EMG. *J Neurophysiol*. 2018;121(1):61-73.
22. Pagan M, Simoncelli EP, Rust NC. Neural quadratic discriminant analysis: Nonlinear decoding with V1-like computation. *Neural Comput*. 2016;28(11):2291-319.
23. Wolpert DH, Macready WG. No free lunch theorems for optimization. *IEEE transactions on evolutionary computation*. 1997;1(1):67-82.

24. Wu W, Black MJ, Gao Y, Serruya M, Shaikhouni A, Donoghue J, et al., editors. Neural decoding of cursor motion using a Kalman filter. *Adv Neural Inf Process Syst*; 2003.
25. Wu W, Hatsopoulos NG. Real-time decoding of nonstationary neural activity in motor cortex. *IEEE Trans Neural Syst Rehabil Eng*. 2008;16(3):213-22.
26. Gilja V, Nuyujukian P, Chestek CA, Cunningham JP, Byron MY, Fan JM, et al. A high-performance neural prosthesis enabled by control algorithm design. *Nat Neurosci*. 2012;15(12):1752.
27. Barbieri R, Wilson MA, Frank LM, Brown EN. An analysis of hippocampal spatio-temporal representations using a Bayesian algorithm for neural spike train decoding. *IEEE Trans Neural Syst Rehabil Eng*. 2005;13(2):131-6.
28. Kloosterman F, Layton SP, Chen Z, Wilson MA. Bayesian decoding using unsorted spikes in the rat hippocampus. *J Neurophysiol*. 2013;111(1):217-27.
29. Friedman J, Hastie T, Tibshirani R. The elements of statistical learning: Springer series in statistics New York; 2001.
30. Liaw A, Wiener M. Classification and regression by randomForest. *R news*. 2002;2(3):18-22.
31. Kaggle. Available from: <https://www.kaggle.com>.
32. Bergstra J, Bengio Y. Random search for hyper-parameter optimization. *Journal of Machine Learning Research*. 2012;13(Feb):281-305.
33. Snoek J, Larochelle H, Adams RP, editors. Practical bayesian optimization of machine learning algorithms. *Adv Neural Inf Process Syst*; 2012.
34. Hyperopt. <http://hyperopt.github.io/hyperopt/>.
35. Pohlmeier EA, Solla SA, Perreault EJ, Miller LE. Prediction of upper limb muscle activity from motor cortical discharge during reaching. *Journal of neural engineering*. 2007;4(4):369.
36. Chang C-C, Lin C-J. LIBSVM: a library for support vector machines. *ACM Transactions on Intelligent Systems and Technology (TIST)*. 2011;2(3):27.
37. Chen T, Guestrin C, editors. Xgboost: A scalable tree boosting system. *Proceedings of the 22Nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*; 2016: ACM.
38. Breiman L. Classification and regression trees: Routledge; 2017.
39. Cho K, Van Merriënboer B, Gulcehre C, Bahdanau D, Bougares F, Schwenk H, et al. Learning phrase representations using RNN encoder-decoder for statistical machine translation. *arXiv preprint arXiv:1406.1078*. 2014.
40. Hochreiter S, Schmidhuber J. Long short-term memory. *Neural Comput*. 1997;9(8):1735-80.
41. Benjamin AS, Fernandes H, Tomlinson T, Ramkumar P, VerSteeg C, Chowdhury R, et al. Modern machine learning as a benchmark for fitting neural responses. *Front Comput Neurosci*. 2018;12:56.
42. Mizuseki K, Sirota A, Pastalkova E, Buzsáki G. Theta oscillations provide temporal windows for local circuit computation in the entorhinal-hippocampal loop. *Neuron*. 2009;64(2):267-80.
43. Mizuseki K, Sirota A, Pastalkova E, Buzsáki G. Multi-unit recordings from the rat hippocampus made during open field foraging. 2009. doi: <http://dx.doi.org/10.6080/K0Z60KZ9>.
44. Kao JC, Stavisky SD, Sussillo D, Nuyujukian P, Shenoy KV. Information systems opportunities in brain-machine interface decoders. *Proceedings of the IEEE*. 2014;102(5):666-82.
45. Nicolas-Alonso LF, Gomez-Gil J. Brain computer interfaces, a review. *Sensors*. 2012;12(2):1211-79.
46. Sussillo D, Stavisky SD, Kao JC, Ryu SI, Shenoy KV. Making brain-machine interfaces robust to future neural variability. *Nature communications*. 2016;7:13749.
47. Farshchian A, Gallego JA, Cohen JP, Bengio Y, Miller LE, Solla SA. Adversarial Domain Adaptation for Stable Brain-Machine Interfaces. *arXiv preprint arXiv:1810.00045*. 2018.
48. Sussillo D, Nuyujukian P, Fan JM, Kao JC, Stavisky SD, Ryu S, et al. A recurrent neural network for closed-loop intracortical brain-machine interface decoders. *Journal of neural engineering*. 2012;9(2):026027.
49. Corbett E, Perreault E, Koerding K, editors. Mixture of time-warped trajectory models for movement decoding. *Adv Neural Inf Process Syst*; 2010.
50. Kao JC, Nuyujukian P, Ryu SI, Shenoy KV. A high-performance neural prosthesis incorporating discrete state selection with hidden Markov models. *IEEE Trans Biomed Eng*. 2017;64(4):935-45.
51. London BM, Miller LE. Responses of somatosensory area 2 neurons to actively and passively generated limb movements. *J Neurophysiol*. 2013;109(6):1505-13.

52. Fagg AH, Ojakangas GW, Miller LE, Hatsopoulos NG. Kinetic trajectory decoding using motor cortical ensembles. *IEEE Trans Neural Syst Rehabil Eng*. 2009;17(5):487-96.
53. Nadeau C, Bengio Y, editors. Inference for the generalization error. *Adv Neural Inf Process Syst*; 2000.
54. Chollet F. Keras. 2015.
55. Glorot X, Bordes A, Bengio Y, editors. Deep sparse rectifier neural networks. *Proceedings of the Fourteenth International Conference on Artificial Intelligence and Statistics*; 2011.
56. Srivastava N, Hinton GE, Krizhevsky A, Sutskever I, Salakhutdinov R. Dropout: a simple way to prevent neural networks from overfitting. *Journal of Machine Learning Research*. 2014;15(1):1929-58.
57. Kingma D, Ba J. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*. 2014.
58. Tieleman T, Hinton G. Lecture 6.5-RmsProp: Divide the gradient by a running average of its recent magnitude. *COURSERA: Neural Networks for Machine Learning*. 2012.
59. Park IM, Archer EW, Priebe N, Pillow JW, editors. Spectral methods for neural characterization using generalized quadratic models. *Adv Neural Inf Process Syst*; 2013.
60. Krizhevsky A, Sutskever I, Hinton GE, editors. Imagenet classification with deep convolutional neural networks. *Adv Neural Inf Process Syst*; 2012.
61. Zhang C, Bengio S, Hardt M, Recht B, Vinyals O. Understanding deep learning requires rethinking generalization. *arXiv preprint arXiv:1611.03530*. 2016.
62. Gao P, Trautmann E, Byron MY, Santhanam G, Ryu S, Shenoy K, et al. A theory of multineuronal dimensionality, dynamics and measurement. *bioRxiv*. 2017:214262.
63. Shenoy KV, Sahani M, Churchland MM. Cortical control of arm movements: a dynamical systems perspective. *Annu Rev Neurosci*. 2013;36:337-59. doi: 10.1146/annurev-neuro-062111-150509. PubMed PMID: 23725001.

Supplemental Information

Demonstration Methods

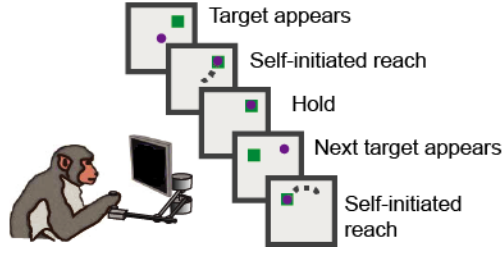
A more complete description of the methods used in our demonstrations follows. While much of this description overlaps with the brief description provided in the main text, we have included the complete description here so that readers interested in the methods details can use this supplement as a primary resource, rather than having to switch between documents.

Demonstration datasets

Decoding movement velocity from the motor and somatosensory cortices: In our “random-target” experiment [19], monkeys moved a planar manipulandum that controlled a cursor on the screen (Fig. S1a). The monkeys continuously reached to newly presented targets, with a very brief hold period (about 200 ms) between reaches. After training, the monkeys were surgically implanted with 96-channel Utah electrode arrays (Blackrock Microsystems, Salt Lake City, UT) to record the extracellular activity of cortical neurons. In one experiment [19], we recorded from both primary motor cortex (M1) and dorsal premotor cortex (PMd) and combined neurons from both areas. The recording from motor cortex was 21 minutes and contained 164 neurons. The mean and median firing rates, respectively, were 6.7 and 3.4 spikes / sec. In another experiment we recorded from area 2 of primary somatosensory cortex (S1) [41]. The recording from S1 was 51 minutes, and contained 52 neurons. The mean and median firing rates, respectively, were 9.3 and 6.3 spikes / sec. From both brain regions, we aimed to predict the x and y components of movement velocity.

Decoding position from the hippocampus: We used a dataset from CRCNS (Collaborative Research in Computational Neuroscience), in which rats chased rewards on a square platform (Fig. S1b) and extracellular recordings were made from layer CA1 of dorsal hippocampus (HC) [42, 43]. More specifically, we used dataset “hc2” and session “ec014.333” from the repository. The recording from HC was 93 minutes, and contained 58 neurons. We did not use the final 20% of the recording, in which the rat had limited movement. We excluded neurons with fewer than 100 spikes over the duration of the experiment, resulting in 46 neurons. The resulting neurons had mean and median firing rates, respectively, of 1.7 and 0.2 spikes / sec. We aimed to predict the x and y position of the rat.

a Task for motor and somatosensory cortex



b Task for hippocampus



Supplemental Figure 1: Tasks and Decoding Schematic

a) In the task for decoding from motor and somatosensory cortices, monkeys moved a planar manipulandum that controlled a cursor on the screen. The monkeys continuously reached to new targets as they were presented, with the exception of a brief hold period between reaches. **b)** In the task for decoding from hippocampus, rats chased rewards on a square platform.

General decoding methods for our demonstration:

Decoding movement velocity from the motor and somatosensory cortices: We predicted the average velocity (x and y components) in 50 ms bins. Neural spike trains used for decoding were also put into 50 ms bins. In motor cortex, we used 700 ms of neural activity (13 bins before and the concurrent bin) to predict the current movement velocity, as a primary interest in the field is investigating how motor cortex affects movement. In somatosensory cortex, we used 650 ms surrounding the movement (6 bins before, the concurrent bin, and 6 bins after), as neural activity has been shown both preceding and following movement [51]. Note that the bins used for decoding differed from those for the Kalman filter and Naïve Bayes decoders (see *Specific Decoders* below). Also, when determining how performance varied as a function of bin size (Fig. S3), we used a slightly different amount of neural data, in order to have a quantity that was divisible by many bin sizes. In this case, for motor cortex, we used 600 ms of neural activity prior to and including the current bin. For somatosensory cortex, we used 600 ms of neural activity centered on the current bin.

Decoding position from the hippocampus: We aimed to predict the position (x and y coordinates) of the rat in 200 ms bins. Neural spike trains used for decoding were also put into 200 ms bins. We used 2 seconds of surrounding neural activity (4 bins before, the concurrent bin, and 5 bins after) to predict the current position. Note that the bins differed from above for the Kalman filter (see *Specific Decoders* below). When determining how performance varied as a function of bin size (Fig. S3), we used 2 seconds of neural activity, centered on the current bin.

Scoring Metric: To determine the goodness of fit, we used

$$R^2 = 1 - \frac{\sum_i (\hat{y}_i - y_i)^2}{\sum_i (\bar{y} - y_i)^2}, \text{ where } \hat{y}_i \text{ are the predicted values, } y_i \text{ are the true values and } \bar{y} \text{ is the mean}$$

value. This formulation of R^2 (which is the fraction of variance accounted for, rather than the squared Pearson's correlation coefficient [52]) can be negative on the test set due to overfitting on

the training set. The reported R^2 values are the average across the x and y components of velocity or position.

Preprocessing: The training input was normalized (z-scored). The training output was zero-centered (mean subtracted), except in support vector regression, where the output was z-scored, which helped algorithm performance. The validation/testing inputs and outputs were preprocessed using the preprocessing parameters from the training set.

Cross-validation: When determining the R^2 for every method (Fig. 3), we used 10 fold cross-validation. For each fold, we split the data into a training set (80% of data), a contiguous validation set (10% of data), and a contiguous testing set (10% of data). For each fold, decoders were trained to minimize the mean squared error between the predicted and true velocities/positions of the training data. We found the algorithm hyperparameters that led to the highest R^2 on the validation set using Bayesian optimization [33]. That is, we fit many models on the training set with different hyperparameters and calculated the R^2 on the validation set. Then, using the hyperparameters that led to the highest validation set R^2 , we calculated the R^2 value on the testing set. Error bars on the test set R^2 values were computed across cross-validation folds. Because the training sets on different folds were overlapping, computing the SEM as σ/\sqrt{n} (where σ is the standard deviation and n is the number of folds) would have underestimated the size of the error bars [53]. We thus calculated the SEM as $\sigma * \sqrt{\frac{1}{n} + \frac{1}{n-1}}$, which takes into account that the estimates across folds are not independent [53].

Bootstrapping: When determining how performance scaled as a function of data size (Figs. 5 and S3), we used single test and validation sets, and varied the amounts of training data that directly preceded the validation set. We did not do this on 10 cross-validation folds due to long run-times. The test and validation sets were 5 minutes long for motor and somatosensory cortices, and 7.5 minutes for hippocampus. To get error bars, we resampled from the test set. Because of the high correlation between temporally adjacent samples, we didn't resample randomly from all examples (which would create highly correlated resamples). Instead, we separated the test set into 20 temporally distinct subsets, S_1 - S_{20} (i.e., S_1 is from $t=1$ to $t=T/20$, S_2 is from $t=T/20$ to $t=2T/20$, etc., where T is the end time), to ensure that the subsets were more nearly independent of each other. We then resampled combinations of these 20 subsets (e.g., $S_5, S_{13}, \dots S_2$) 1000 times to get confidence intervals of R^2 values.

Implementation details for specific decoders:

Wiener Filter: The Wiener filter uses multiple linear regression to predict the output from multiple time bins of every neurons' spikes. That is, the output is assumed to be a linear mapping of the number of spikes in the relevant time bins from every neuron (Fig. 1a,b). We used separate models to predict the x and y components of the kinematics.

Wiener Cascade: The Wiener cascade (also known as a linear-nonlinear model) fits a linear regression (the Wiener filter) followed by a fitted static nonlinearity (e.g., [35]). This allows for a nonlinear relationship between the input and the output, and assumes that this nonlinearity is purely a function of the linear output. Here, as in the Wiener Filter, the input was neurons' spike

rates over relevant time bins. The nonlinear component was a polynomial with degree determined on the validation set. Separate models were used to predict the x and y components of the kinematics.

Support Vector Regression: In support vector machine regression (SVR) [36], the inputs are projected into a higher-dimensional space using a nonlinear kernel, and then linearly mapped from this space to the output to minimize an objective function [36]. Here, we used standard support vector regression (SVR) with a radial basis function kernel to predict the kinematics from the neurons' spike rates in each bin. We set hyperparameters for the penalty of the error term and the maximum number of iterations. Separate models were used to predict the x and y components of the kinematics.

XGBoost: XGBoost (Extreme Gradient Boosting) [37] is an implementation of gradient boosted trees. Tree-based methods sequentially split the input space into many discrete parts (visualized as branches on a tree for each split), in order to assign each final "leaf" (a portion of input space that is not split any more) a value in output space [38]. We fit many regression trees, which are trees that predict continuous output values. "Gradient boosting" refers to fitting each subsequent regression tree to the residuals of the previous fit. Here, we used XGBoost to predict the kinematics from the neurons' spike rates in each bin. We set hyperparameters for the maximum depth of the tree, number of trees, and learning rate. We fit separate models to predict the x and y components.

Feedforward Neural Network: A feedforward neural net connects the inputs to sequential layers of hidden units, which then connect to the output. Each layer connects to the next (e.g., the input layer to the first hidden layer, or the first to second hidden layers) via linear mappings followed by nonlinearities. Note that the Wiener cascade is a special case of a neural network with no hidden layers. Using the Keras library [54], we created a fully connected (dense) feedforward neural network with 2 hidden layers and rectified linear unit [55] activations after each hidden layer. We required the number of hidden units in each layer to be the same. We set hyperparameters for the number of hidden units in the layers, amount of dropout [56], and number of training epochs. We used the Adam algorithm [57] as the optimization routine. This neural network, and all neural networks below had two output units. That is, the same network predicted the x and y components together, rather than separately. The input was still the number of spikes in each bin from every neuron. Note that we refer to feedforward neural networks as a "modern" technique, despite their having been used for many decades, due to their current resurgence and the modern methods for training.

Simple RNN: In a standard recurrent neural network (RNN), the hidden state is a linear combination of the inputs and the previous hidden state. This hidden state is then run through an output nonlinearity, and linearly mapped to the output. RNNs, unlike feedforward neural networks, allow temporal changes in the system to be modeled explicitly. Here, using the Keras library [54], we created a neural network architecture in which the spiking inputs from all neurons were fed into a standard recurrent neural network (Fig. 1c). The units from this recurrent layer were fed through rectified linear unit nonlinearities, and fully connected to an output layer with two units (x and y velocity or position components). We set hyperparameters for the number of units, amount of dropout, and number of training epochs. We used RMSprop [58] as the optimization routine.

Gated Recurrent Unit: Gated recurrent units (GRUs) [39] are a more complex type of recurrent neural network. It has gated units, which in practice allow for better learning of long-term dependencies. Almost all implementation methods were the same as for the simple RNN, except Gated Recurrent Units were used instead. An implementation difference is that the units from the recurrent layers were fed through hyperbolic tangent (tanh) activations (as is standard for GRUs) rather than rectified linear unit activations.

Long Short Term Memory Network: Like the GRU, the long short term memory (LSTM) network [40] is a more complex recurrent neural network with gated units that further improve the capture of long-term dependencies. The LSTM has more parameters than the GRU. All implementation methods were the same as for GRUs, except LSTM units were used instead.

Ensemble: Ensemble techniques combine the predictions from several methods, and thus have the potential to leverage their different benefits. We used the predictions from all decoders except the Kalman filter and Naïve Bayes decoders (which have different formats). We combined the predictions from the above 8 methods using a feedforward neural network. That is, the 8 methods' predictions were provided as input into a feedforward neural network that we trained to predict the true output. This was done separately for the x and y components of the position or velocity.

Kalman Filter: Our Kalman filter for neural decoding was based on [24]. As our implementation is not identical, we write out the full description here. In the Kalman filter, the hidden state at time t is a linear function of the hidden state at time $t-1$, plus a matrix characterizing the uncertainty. For neural decoding, the hidden state is the kinematics (x and y components of position, velocity, and acceleration). Note that even though we only aim to predict position or velocity, all kinematics are included because this allows for better prediction. More formally,

$$\mathbf{y}_t = \mathbf{A}\mathbf{y}_{t-1} + \mathbf{w}$$

where \mathbf{y}_t and \mathbf{y}_{t-1} are 6 x 1 vectors, \mathbf{A} is a 6 x 6 matrix, and \mathbf{w} is sampled from a normal distribution, $N(0, \mathbf{W})$, with mean 0 and covariance \mathbf{W} . \mathbf{W} is the 6 x 6 uncertainty matrix.

The observation (measurement) at time t^* is a linear function of the hidden state at time t (plus noise). For neural decoding, the measurement is the neural activity. Note that we allowed a lag between the neural data and predicted kinematics, which is why we use t^* for the time of the neural activity. More formally,

$$\mathbf{x}_{t^*} = \mathbf{H}\mathbf{y}_t + \mathbf{q}$$

where \mathbf{x}_{t^*} is an N x 1 vector, \mathbf{H} is an N x 6 matrix, and \mathbf{q} is sampled from a normal distribution, $N(0, \mathbf{Q})$, with mean 0 and covariance \mathbf{Q} . \mathbf{Q} is the N x N measurement noise matrix.

During training, \mathbf{A} , \mathbf{H} , \mathbf{W} , and \mathbf{Q} 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 [24] or our code for the update equations (note that \mathbf{x} and \mathbf{y} have different notation in [24]).

We had one hyperparameter which differed from the standard implementation [24]. We divided the noise matrix associated with the transition in kinematic states, \mathbf{W} , by the hyperparameter scalar C , 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 (Fig. S4). We also allowed for a lag between the neural data and predicted kinematics. The lag and hyperparameter were determined based on validation set performance.

Naïve Bayes: The Naïve Bayes decoder is a type of Bayesian decoder that determines the probabilities of different outcomes, and it then predicts the most probable. Briefly, it fits an encoding model to each neuron, makes conditional independence assumptions about neurons, and then uses Bayes' rule to create a decoding model from the encoding models. This probabilistic framework can incorporate prior information about the variables.

We used a Naïve Bayes decoder similar to the one implemented in [5]. As with the Kalman filter, because this is not an out-of-the-box method that we applied, we write the full details here. We first fit an encoding model (tuning curve) using the output variables. Let $f_i(\mathbf{s})$ be the value of the tuning curve (the expected number of spikes) for neuron i at the output variables \mathbf{s} . Note that \mathbf{s} is a vector containing the two output variables we are predicting (x and y positions/velocities). We assume the number of recorded spikes in the given bin, r_i , is generated from the tuning curve with Poisson statistics:

$$P(r_i|\mathbf{s}) = \frac{\exp[-f_i(\mathbf{s})]f_i(\mathbf{s})^{r_i}}{r_i!}$$

We also assume that all the neurons' spike counts are conditionally independent given the output variables, so that:

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

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.

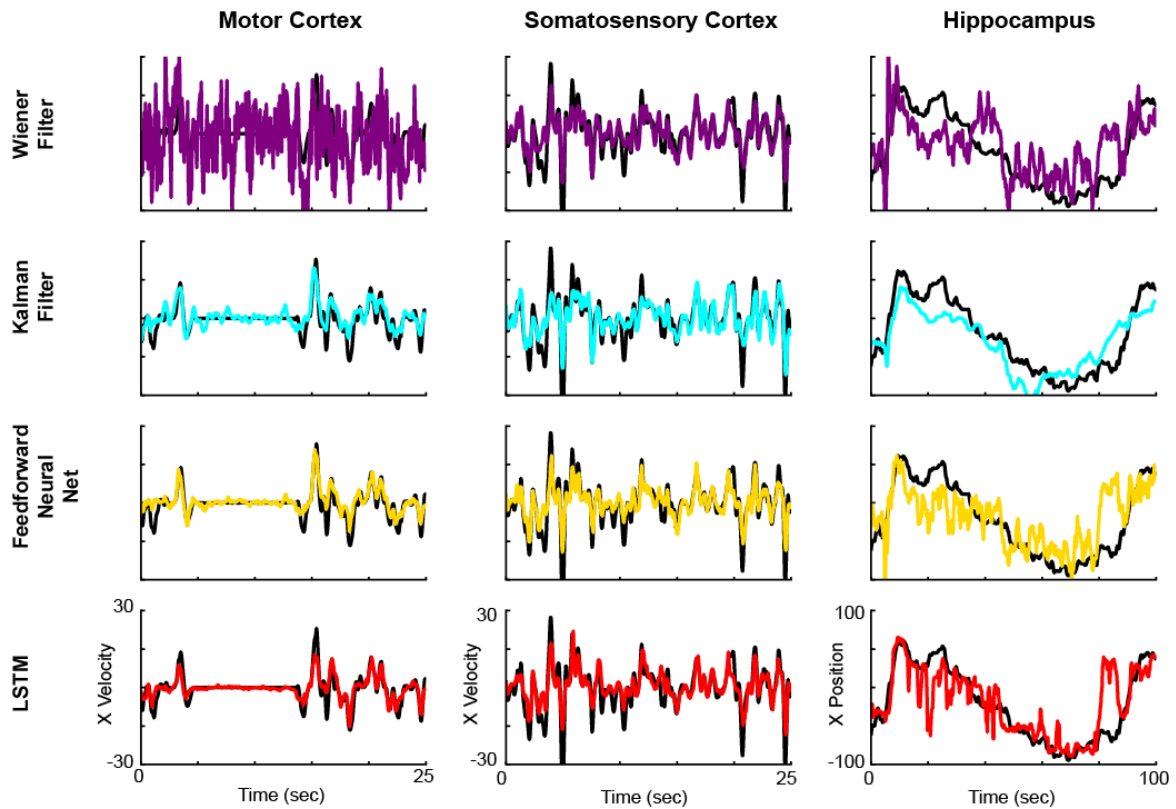
Additionally, including $P(\mathbf{s})$ based on the distribution of output variables in the training set did not improve performance on the validation set. This could be because the probability distribution differed between the training and validation/testing sets, or because the distribution of output 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 [59], 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 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.

Additional Demonstration Results

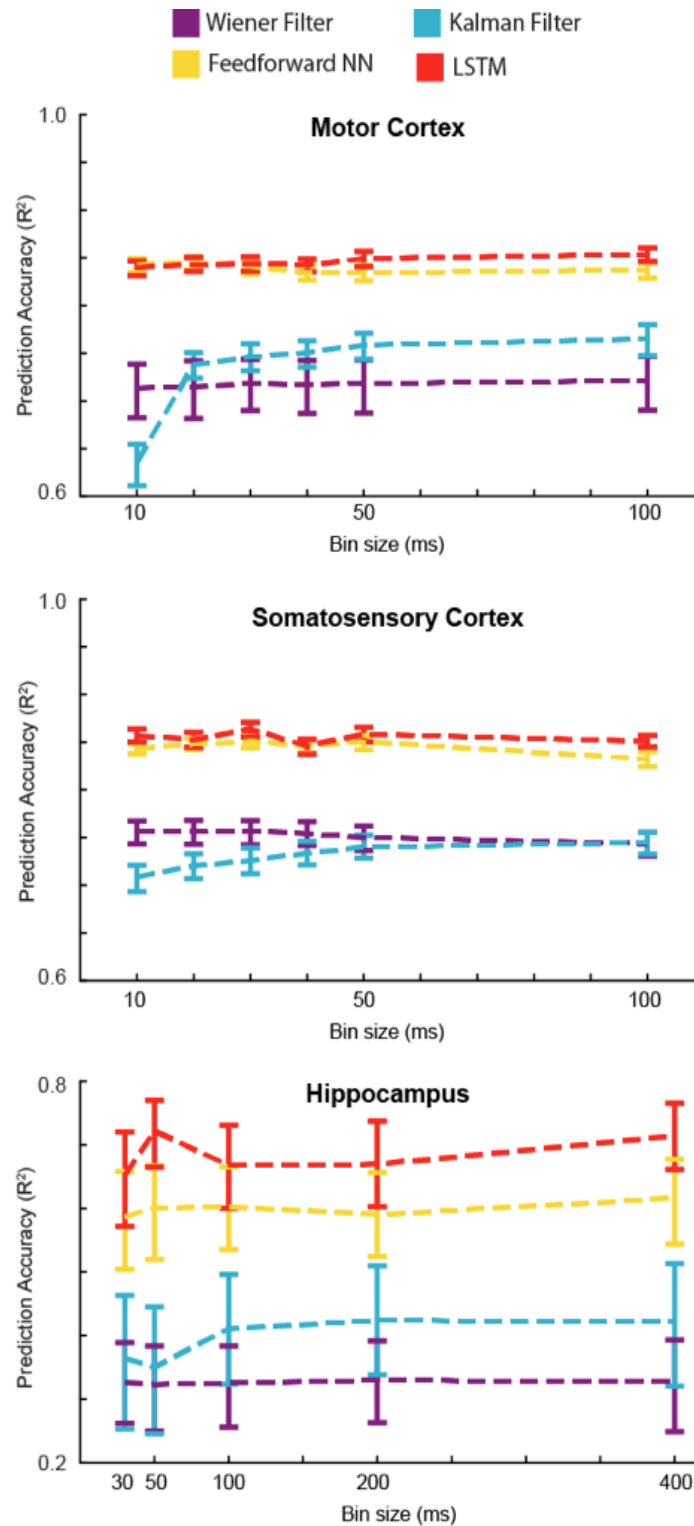


Supplemental Figure 2: Example results with limited training data

Using only 2 minutes of training data for motor cortex and somatosensory cortex, and 15 minutes of training data for hippocampus, we trained two traditional methods (Wiener filter and Kalman filter), and two modern methods (feedforward neural network and LSTM). Example decoding results are shown from motor cortex (left), somatosensory cortex (middle), and hippocampus (right), for these methods (top to bottom). Ground truth traces are in black, while decoder results are in the same colors as previous figures.

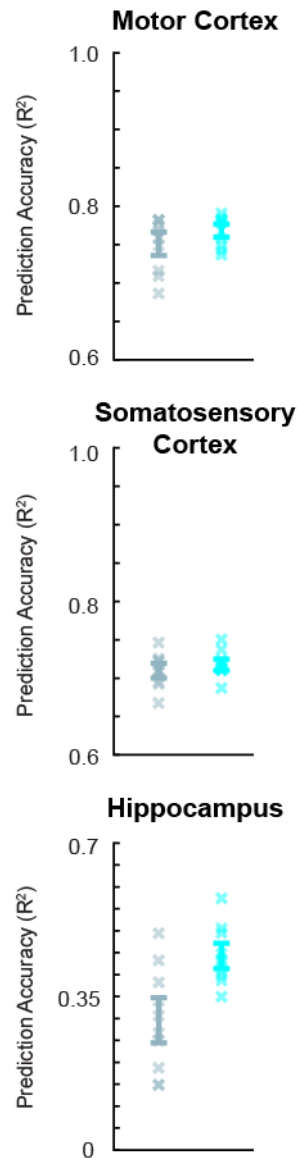
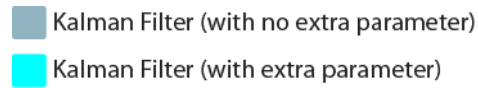
Varying bin sizes

While all demonstrations in the main text only tested decoding accuracy using a set bin size, different decoding applications may require different temporal resolutions. It is therefore also important to compare decoding accuracy with varying bin sizes. For the motor and somatosensory cortex datasets, we tested bin sizes from 10-100 ms. For the hippocampus dataset, we tested bin sizes from 30-400 ms. Surprisingly, for all methods besides the Kalman filter, decoding performance remained consistent across all the tested bin sizes (Fig. S3). For the Kalman filter, decoding performance increased as bin sizes became larger, likely because the Kalman filter used a single bin of neural data to make predictions (unlike the other methods that used many bins), and small bin sizes could thus lead to insufficient neural data to predict behavior well. Importantly, for all bin sizes, the modern methods had better decoding accuracy. Thus, modern machine learning methods remain advantageous regardless of the temporal resolution.



Supplemental Figure 3: Decoder results with different bin sizes

Using varying bin sizes, we trained two traditional methods (Wiener filter and Kalman filter), and two modern methods (feedforward neural network and LSTM). We used the same testing set as in Fig. 5, and the largest training set from Fig. 5. R^2 values are reported for these decoders (different colors) for each brain area (top to bottom). Error bars are 68% confidence intervals (meant to approximate the SEM) produced via bootstrapping, as we used a single test set.



Supplemental Figure 4. Kalman Filter Versions

R^2 values are reported for different versions of the Kalman Filter for each brain area (top to bottom). On the left (in bluish gray), the Kalman Filter is implemented as in [24]. On the right (in cyan), the Kalman Filter is implemented with an extra parameter that scales the noise matrix associated with the transition in kinematic states (see *Demonstration Methods*). This version with the extra parameter is the one used in the main text. Error bars represent the mean \pm SEM across cross-validation folds. X's represent the R^2 values of each cross-validation fold. Note the different y-axis limits for the hippocampus dataset.

Additional Discussion of Main-text Demonstrations

We find it particularly interesting that the neural network methods worked so well with limited data, which is counter to the common perception. We believe the explanation is simply the size of the networks. For instance, our networks have on the order of 10^5 parameters, while common networks for image classification (e.g., [60]) can have on the order of 10^8 parameters. Thus, the smaller size of our networks (hundreds of hidden units) may have allowed for excellent prediction with limited data [61]. Moreover, the fact that the tasks we used had a low-dimensional structure, and therefore the neural data was also likely low dimensional [62], might allow high decoding performance with limited data.

It is also intriguing that the feedforward neural network did almost as well as the LSTM and better than the standard RNN, considering the recent attention to treating the brain as a dynamical system [63]. For the motor and somatosensory cortex decoding, it is possible that the highly trained monkeys yielded a stereotyped temporal relationship between neural activity and movement that a feedforward neural network could effectively capture. It would be interesting to compare the performance of feedforward and recurrent neural networks on less constrained behavior.

In order to find the best hyperparameters for the decoding algorithms, we used a Bayesian optimization routine [33] to search the hyperparameter space (see *Demonstration Methods*). Still, it is possible that, for some of the decoding algorithms, the hyperparameters were nonoptimal, which would have lowered overall accuracy. Moreover, for several methods, we did not attempt to optimize all the hyperparameters. We did this in order to simplify the use of the methods, decrease computational runtime during hyperparameter optimization, and because optimizing additional hyperparameters (beyond default values) did not appear to improve accuracy. For example, for the neural nets we used dropout but not L1 or L2 regularization, and for XGBoost we optimized less than half the available hyperparameters designed to avoid overfitting. While our preliminary testing with additional hyperparameters did not appear to change the results significantly, it is possible that some methods did not achieve optimal performance.