RESCIENCE C

Replication / Computational Neuroscience

[Re] The Discriminative Kalman Filter for Bayesian Filtering with Nonlinear and Non-Gaussian Observation Models

Josue Casco-Rodriguez[1], Caleb Kemere[1], and Richard G. Baraniuk[1]
[1]Rice University, Houston, Texas, USA

Edited by
Benoît Girard

Reviewed by
Ozan Caglayan

Received
19 June 2023

Published
08 April 2025

DOI
10.5281/zenodo.15172014

Abstract

Kalman filters provide a straightforward and interpretable means to estimate hidden or latent variables, and have found numerous applications in control, robotics, signal processing, and machine learning. One such application is neural decoding for neuroprostheses. In 2020, Burkhart et al. thoroughly evaluated their new version of the Kalman filter that leverages Bayes' theorem to improve filter performance for highly non-linear or non-Gaussian observation models. This work provides an open-source Python alternative to the authors' MATLAB algorithm. Specifically, we reproduce their most salient results for neuroscientific contexts and further examine the efficacy of their filter using multiple random seeds and previously unused trials from the authors' dataset. All experiments were performed offline on a single computer.

1 Introduction

Brain-computer interfaces (BCIs) have long been a subject of science fiction [1]. Detailed communication with a machine through mere thought has become more technologically feasible with time, but still remains infeasible for the general public. However, for certain groups of people, BCIs are a necessary means to circumvent debilitating circumstances. For example, people experiencing quadriplegia or locked-in syndrome have very little means through which to communicate or interact with the outside world, and thus stand to benefit from thought-controlled interfaces through which they can operate robotic limbs or computers [1, 2]. As another example, people with impaired control or loss of a limb also benefit from robotic prosthetic limbs that can be controlled through thought alone [3]. In the aforementioned applications of BCIs, one of the key algorithmic challenges is to accurately estimate some relevant aspect of the user's cognition. Specifically, BCIs and neuroprosthetics often seek to decode a quantifiable motor intention signal that can be used to control robots or cursors, such as the velocity of an intended arm, hand, or finger movement [1, 2]. Such neural decodings are usually made using information from a subset of the user's neurons, made accessible through invasive electrode technologies or through other non-invasive means [4].

The Kalman filter [5] is a common basis upon which practicioners develop BCI decoding methods [6, 7, 8]. Burkhart et al. (2020) [9] sought to improve Kalman filter performance

Correspondence should be addressed to Josue Casco-Rodriguez (jc135@rice.edu)
The authors have declared that no competing interests exist.
Code is available at https://github.com/Josuelmet/Discriminative-Kalman-Filter-4.5-Python. – SWH swh:1:dir:859cfa9e3d94c2aa580211c9de5d7a28d43a7fe2.
Data is available at https://portal.nersc.gov/project/crcns/download/dream/data_sets/Flint_2012 – DOI 10.1088/1741-2560/9/4/046006.
Open peer review is available at https://github.com/ReScience/submissions/issues/74.

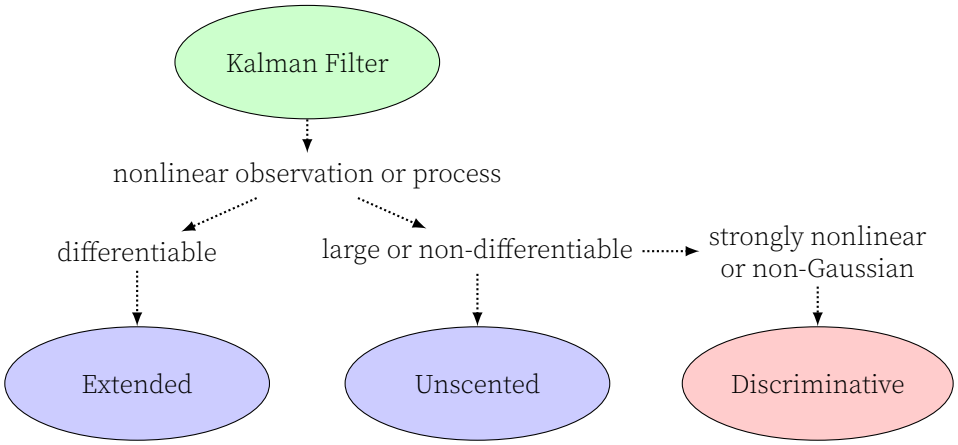

Figure 1. Summary of how the observation and process models of various Bayesian filtering methods relax those of the Kalman filter.

in neural decoding efforts by developing the Discriminative Kalman Filter (DKF), which leverages Bayes's theorem to facilitate Bayesian filtering in contexts involving highly non-linear or non-Gaussian observations. The authors presented five different experiments verifying the efficacy of the DKF: the first three experiments (4.2–4.4) consisted of intricate toy examples with known observation models, while the final two were BCI-focused experiments with observations consisting of neural recordings. The last two experiments (4.5 and 4.6) are the most salient to BCI applications because the mapping from neural activity to thoughts or intentions is highly nonlinear [10] and usually unknown to practicioners. We chose to solely replicate Experiment 4.5 because it was the most BCI-oriented experiment whose data and code were publicly available, since Experiment 4.6 involved human data.

2 Kalman Filter Variations

2.1 Kalman Filter

Kalman filters [5] are a family of algorithms whose purpose is to estimate a set of unobservable latent states $\{\boldsymbol{Z}_1, \boldsymbol{Z}_2, ..., \boldsymbol{Z}_T\}$ given a set of observations $\{\boldsymbol{X}_1, \boldsymbol{X}_2, ..., \boldsymbol{X}_T\}$. Kalman filters operate under the Markov assumption: any observation $\boldsymbol{X}_i$ depends only on its corresponding state $\boldsymbol{Z}_i$, and any state $\boldsymbol{Z}_i$ depends only on the immediately preceding state $\boldsymbol{Z}_{i-1}$. Unlike Hidden Markov models [11], which also operate under the Markov assumption, the latent states $\{\boldsymbol{Z}_i\}_{i=1}^{T}$ are not discrete, instead having continuous values. For real-time neural decoding applications, the primary role of Kalman filters is to predict the latest latent state (e.g., a finger velocity) $\boldsymbol{Z}_T$ when given the previous latent state $\boldsymbol{Z}_{T-1}$ and the latest observation (e.g., neural electrode signals) $\boldsymbol{X}_T$. Outside of real-time control, other applications of Kalman filters include smoothing (predicting $\boldsymbol{Z}_i$ for any $1 \leq i \leq T$ when given $\{\boldsymbol{X}_i\}_{i=1}^{T}$) and projection into the future (predicting $\{\boldsymbol{Z}_i\}_{i=T}^{\infty}$ given $\{\boldsymbol{X}_i\}_{i=1}^{T}$).

The original Kalman Filter (KF) is a linear, Gaussian, and stationary model, and thus assumes the following:

1. Linear Gaussian observations: $p(\boldsymbol{X}_i|\boldsymbol{Z}_i) \sim \mathcal{N}(\boldsymbol{H}\boldsymbol{Z}_i, \boldsymbol{R})$
2. Linear Gaussian latent dynamics: $p(\boldsymbol{Z}_i|\boldsymbol{Z}_{i-1}) \sim \mathcal{N}(\boldsymbol{A}\boldsymbol{Z}_{i-1}, \boldsymbol{G})$[1]

[1] Some practitioners prefer to write $\boldsymbol{A}$ as $\boldsymbol{F}$, and $\boldsymbol{G}$ as $\boldsymbol{Q}$.

2.2 Extended and Unscented Kalman Filters

The KF is the optimal estimator for linear dynamic systems with Gaussian observation and process noise, but neural processing systems are usually highly nonlinear [10]. The simplest modification for handling nonlinear observation models is the Extended Kalman Filter (EKF) [12], which makes the following modifications to the original KF model:

1. Nonlinear differentiable Gaussian observations: $p(\boldsymbol{Z}_i|\boldsymbol{X}_i) \sim \mathcal{N}\left(f^{-1}(\boldsymbol{X}_i), \boldsymbol{R}\right)$
2. Nonlinear differentiable Gaussian latent dynamics: $p(\boldsymbol{Z}_i|\boldsymbol{Z}_{i-1}) \sim \mathcal{N}\left(h(\boldsymbol{Z}_{i-1}), \boldsymbol{G}\right)$
3. The observation and process transformations $f^{-1}(\cdot)$ and $h(\cdot)$ cannot be applied directly to the latent state covariance. Instead, their Jacobians evaluated at the current latent value $\boldsymbol{Z}_i$ are applied to the latent covariance at timestep i.

While the EKF can perform well with sufficient knowledge of the system, it can also perform poorly without such knowledge, or when strong nonlinearities are involved in the system. Another nonlinear Kalman filter algorithm is the Unscented Kalman Filter (UKF) [13]. The UKF differs from the EKF by using a deterministic sampling technique known as the unscented transform to pick a minimal set of sample points (sigma points) around the mean. By incorporating sampling techniques, the UKF allows usage of transformations whose Jacobians are difficult or impossible to calculate (i.e., large or non-differentiable functions).

2.3 Discriminative Kalman Filter

The Discriminative Kalman Filter (DKF) [14] keeps the following model assumptions from the Kalman Filter:

1. Linear Gaussian latent dynamics: $p(\boldsymbol{Z}_i|\boldsymbol{Z}_{i-1}) \sim \mathcal{N}\left(\boldsymbol{A}\boldsymbol{Z}_{i-1}, \boldsymbol{G}\right)$
2. The observation model $p(\boldsymbol{X}_i|\boldsymbol{Z}_i)$ is stationary.

However, unlike the Kalman Filter, the DKF does not assume a linear observation model $p(\boldsymbol{X}_i|\boldsymbol{Z}_i)$. Instead, the DKF approximates the observation model $p(\boldsymbol{X}_i|\boldsymbol{Z}_i) \approx p(\boldsymbol{Z}_i|\boldsymbol{X}_i)/p(\boldsymbol{Z}_i)$ via Bayes' theorem. Such a substitution can prove useful if (a) the observation distribution $p(\boldsymbol{X}_i|\boldsymbol{Z}_i)$ is strongly nonlinear or non-Gaussian, or (b) the dimensionality of observations $\boldsymbol{X}$ is much larger than the dimensionality of latents $\boldsymbol{Z}$; in neural signal processing, both are often true. The DKF further models $p(\boldsymbol{Z}_i|\boldsymbol{X}_i)$ as Gaussian: $p(\boldsymbol{Z}_i|\boldsymbol{X}_i) \sim \mathcal{N}(f(\boldsymbol{X}_i), Q(\boldsymbol{X}_i))$, where $f(\cdot)$ and $Q(\cdot)$ are nonlinear functions that map the observation $\boldsymbol{X}_i$ to its corresponding elements in the state space $\mathbb{R}^d$ and the covariance space $\mathbb{S}^d$, respectively. Burkhart et al. (2020) [9] formulate that the observation-to-state transformation $f(\cdot)$ and the observation-to-covariance transformation $Q(\cdot)$ are the conditional mean and covariance of $\boldsymbol{Z}_i$ given $\boldsymbol{X}_i$:

$$f(\boldsymbol{X}_i) = \mathbb{E}(\boldsymbol{Z} \mid \boldsymbol{X} = \boldsymbol{X}_i) \tag{1}$$

$$Q(\boldsymbol{X}_i) = \mathrm{Var}(\boldsymbol{Z} \mid \boldsymbol{X} = \boldsymbol{X}_i) \tag{2}$$

The DKF also makes the stationarity assumption $p(\boldsymbol{Z}_i) \sim \mathcal{N}(\boldsymbol{0}, \boldsymbol{S})$, where $\boldsymbol{S}$ (also written as $\boldsymbol{V_Z}$ or $\boldsymbol{T}$) is the covariance of $\boldsymbol{Z}_i$ when not conditioned on any $\boldsymbol{Z}_{i-1}$. Defining the initial latent state estimate $\boldsymbol{\mu}_0 = \boldsymbol{0}$ and the initial latent covariance estimate $\boldsymbol{\Sigma}_0 = \boldsymbol{S}$ (the latent covariance $\boldsymbol{\Sigma}$ is also written as $\boldsymbol{P}$ in traditional filtering literature), each iteration of the DKF algorithm proceeds as follows:

$$\boldsymbol{v}_i = \boldsymbol{A}\boldsymbol{\mu}_{i-1} \tag{3}$$

$$\boldsymbol{M}_i = \boldsymbol{A}\boldsymbol{\Sigma}_{i-1}\boldsymbol{A}^T + \boldsymbol{G} \tag{4}$$

$$\boldsymbol{\Sigma}_i = \left(\boldsymbol{M}_i^{-1} + Q(\boldsymbol{X}_i)^{-1} - \boldsymbol{S}^{-1}\right)^{-1} \tag{5}$$

$$\boldsymbol{\mu}_i = \boldsymbol{\Sigma}_i \left(\boldsymbol{M}_i^{-1}\boldsymbol{v}_i + Q(\boldsymbol{x}_i)^{-1} f(\boldsymbol{X}_i)\right) \tag{6}$$

In practice, the following additional changes are made to the DKF algorithm, with pseudo-inverses written as †:

1. $Q(\boldsymbol{X}_i)^{-1} - \boldsymbol{S}^{-1}$ must be positive definite. If it is not, set $Q(\boldsymbol{X}_i)^{-1} = \left(Q(\boldsymbol{X}_i)^{-1} + \boldsymbol{S}^{-1}\right)^{-1}$.
2. $\boldsymbol{\Sigma}_i = \left(\boldsymbol{M}_i^\dagger + Q(\boldsymbol{X}_i)^\dagger - \boldsymbol{S}^\dagger\right)^\dagger$
3. $\boldsymbol{\mu}_i = \boldsymbol{\Sigma}_i \left(\boldsymbol{M}_i^\dagger \boldsymbol{v}_i + Q(\boldsymbol{X}_i)^\dagger f(\boldsymbol{X}_i)\right)$

Another formulation of the DKF more suitable for real-time applications is the Robust DKF, which makes the assumption that the eigenvalues of $\boldsymbol{S}^{-1}$ are so small that the $-\boldsymbol{S}^{-1}$ term in Equation 5 is negligible and can thus be removed. Additionally, the Robust DKF places an improper prior on $\boldsymbol{Z}_0$, and modeling starts from $t = 1$. Specifically, $\boldsymbol{\mu}_1 = f(\boldsymbol{X}_1)$ and $\boldsymbol{\Sigma}_1 = Q(\boldsymbol{X}_1)$, and iterative calculations start at $t = 2$ instead of $t = 1$.

3 Methods

3.1 Data

The data that Burkhart et al. (2020) [9] used in their most salient and practical open-source experiment for neuroscientific applications (4.5) came from Flint et al. (2012) [15]. Specifically, the data is from a 96-channel microelectrode array implanted in the primary motor cortex of one rhesus macaque (Monkey C, center-out from [15]). The macaque was taught to earn juice rewards by moving a manipulandum in a center-out reaching task. A 128-channel acquisition system recorded the resulting signals, which were sampled at 30 kHz, highpass-filtered at 300 Hz, and then thresholded and sorted into spikes offline. The data is made publicly available by Walker and Kording (2013) [16] under the Database for Reaching Experiments and Models (DREAM)[2]

3.2 Preprocessing

The five *.mat* files from the Flint et al. (2012) manipulandum manipulation trials [15] are first placed in the same directory as *flint_preprocess_data.ipynb* for organization and preprocessing. Each file has various recording sessions (henceforth referred to as *trials*), where each session (trial) in a given file was recorded on the same day [15]. Since the trial data is stored as MATLAB structs, extra attention had to be paid during implementation to ensure that the Python data organization matched the original MATLAB organization method. Each of the five files are processed as follows:

1. Isolate all data from the n-th trial of the file (some files only have one trial, while other file have up to four trials).
2. For each trial in the file:
   a) Stack each timestamp's 3D manipulandum velocity vertically.
   b) Discretize the activity of each neuron in the trial into bins of $\frac{1}{30\text{kHz}} \approx 33\,\mu\text{s}$. Stack the neurons' spike bins vertically.

[2] Data is available at https://portal.nersc.gov/project/crcns/download/dream/data_sets/Flint_2012

3. Only keep the first two dimensions of the velocities, since the third dimension is not relevant for operation of the manipulandum.
4. Save the resulting 2D array of manipulandum velocities from the i-th trial as a unique entry in a list of 2D velocity arrays. In equivalent fashion, store the array of spike bins in a list of 2D bin arrays.
5. Repeat all earlier steps for each trial from each file. There are a total of 12 trials across all files, treating each trial as independent from the others.

Once organized, the lists of trial velocities and spike bins are preprocessed in the same manner as Burkhart et al. (2020) [9]:

1. Isolate the velocities and spike bins from the n-th trial.
2. Downsample the spike bin samples from 33 µs intervals to 100 ms intervals.
3. Replace the spike bin data with a moving sum (with a window length of 10 entries) of the spike bin data.
4. Downsample the velocity data samples into intervals of 100 ms—specifically, keep entries whose indices are halfway between the indices that were used to downsample the spike bin data earlier.
5. Replace the spike bin data with the z-scores (using 1 degree of freedom correction, as per MATLAB's zscore function) of its top 10 principal components.
6. As during organization, store the 2D array of processed spike bins and the 2D array of processed velocities in a list of 2D spike arrays and a list of 2D velocity arrays, respectively.
7. Repeat all previous steps for each of the 12 trials.

In similar fashion as Burkhart et al. (2020) [9], data preprocessing yields an array of processed velocities[3] and an array of processed spike bins[4]. The resulting latent states (manipulandum velocities) are 2-dimensional, while the observations (z-scored principal component scores of neural activity) are 10-dimensional.

3.3 Computation

Paper_Script_45.ipynb reproduces Experiment 4.5 from Burkhart et al. (2020) [9] in Python, based on the authors' original MATLAB implementation. While there are a total of 12 trials, the authors only conducted analysis on the first 6 trials' data. The following analyses and procedures were performed separately on each of the aforementioned trials. See Table 1 for an overview of the similarities and differences between our computational implementations and those of Burkhart et al. (2020) [9].

Training, Validation, and Test Data Split — Before performing any regression, the data from the current trial is isolated and split into training and test data. Recall that each trial represents a recorded session of a macaque center-out reaching task [15], and has several thousand 100 ms samples (approximately 10 minutes of recorded data) [15]. The first 5,000 samples are used as training data, while the subsequent 1,000 samples are used as test data. Various methods are used for learning either the observation-to-state transformation $f : \mathbb{R}^{10} \to \mathbb{R}^2$ or the state-to-observation transformation $f^{-1} : \mathbb{R}^2 \to \mathbb{R}^{10}$.

[3] https://github.com/Josuelmet/Discriminative-Kalman-Filter-4.5-Python/blob/main/flint-data-preprocessing/procd_velocities.npy

[4] https://github.com/Josuelmet/Discriminative-Kalman-Filter-4.5-Python/blob/main/flint-data-preprocessing/procd_spikes.npy

Algorithm	Burkhart et al. (2020)	Python Reproduction
Preprocessing	See Section 3.2	Same, to the best of our ability
Kalman Filter	Fully deterministic matrix implementation	Same
Neural Network	One hidden layer of 10 tanh neurons optimizied via Bayesian-regularized Levenberg-Marquardt method	Two hidden layers of 10 tanh neurons optimizied via RMSProp with $l2$ weight penalty and 4,000 epochs
Nadaraya-Watson	Kernel bandwidth optimization via the MATLAB fminunc function	Kernel bandwidth optimization via the scipy minimize_scalar function
Gaussian Process	Fitting via GPML package with RBF kernel	Fitting via scikitlearn GaussianProcessRegressor with RBF kernel
Long Short-Term Memory	One 20-dimensional LSTM layer and one fully-connected layer optimized via AdaGrad with dropout	One 20-dimensional LSTM layer and one fully-connected layer optimized via Adam with an $l2$ weight penalty and no dropout
Extended and Unscented Kalman Filters	Native MATLAB implementation using the earlier one-layer neural network architecture	FilterPy implementation using the earlier two-layer neural network architecture

Table 1. The summarized differences between the algorithmic implementations in Burkhart et al. (2020) [9] and this study.

However, only Nadaraya-Watson (NW) kernel regression [17] is used for learning the transformation $Q : \mathbb{R}^{10} \rightarrow \mathbb{S}^2$ that estimates the conditional covariance of the latent $\boldsymbol{Z}_i$ given the observation $\boldsymbol{X}_i$. Following Burkhart et al. (2020) [9], we used 70% of the training data (3,500 samples) purely to train regression models, while the remaining 30% (1,500 samples) are used to learn Q(x) using NW regression; thus. The aforementioned 3,5000 samples will be referred to as training data, the next 1,500 samples will be referred to as validation data, and the last 1,000 samples will be referred to as testing data, for clarity. While the indices of the 5,000 (training + validation) and 1,000 (testing) samples are not randomized due to the temporally sensitive nature of BCI decoding, the indices of the 3,500 (training) and the 1,500 (validation) samples are randomly drawn (without replacement) from the 5,000 samples.

Linear Kalman Filter — The first regression method is the fundamental baseline upon which to compare all subsequent algorithms: the traditional Kalman filter. It uses all 5,000 training and validation samples as training data, since it does not need a validation set with which to learn a Q(x) covariance function. Aside from estimated latent state means and covariances, the Kalman filter also yields the transition matrix $\boldsymbol{A}$, the process noise covariance $\boldsymbol{Q}$ (named $\boldsymbol{G}$ in the notebook), and the initial estimate covariance $\boldsymbol{V_Z}$ (also referred to as $\boldsymbol{V}_0$ or $\boldsymbol{P}_0$).

Neural Network Regression — The next regression method is the Discriminative Kalman Filter (DKF) using neural network (NN) regression. Recall that all DKF methods must learn the observation-to-state transformation $f : \mathbb{R}^{10} \to \mathbb{R}^2$. Neural network regression estimates $f(\cdot)$ with a feedforward network that learns from the training data (3,500 samples).

Burkhart et al. (2020) [9] used a neural network with one hidden layer of 10 hyperbolic tangent neurons. However, the authors trained their smaller network using a combination of Levenberg-Marquardt (LM) optimization algorithm and Bayesian regularization (BR) [18], which automatically calculate $l2$ weight penalties iteratively and incorporate information from the inverse Hessian of the loss function. However, since resources for second-order optimization are scarce in Python, we used a more traditional neural network architecture—2 hidden layers of 10 hyperbolic tangent neurons each—and optimization method—4,000 epochs using RMSProp [19] with a learning rate of 10^{-3} and an $l2$ regularization penalty of 10^{-4}.

After estimating the function $f : \mathbb{R}^{10} \to \mathbb{R}^2$, the network predicts the latent states (velocities) corresponding to the validation data observations (1,500 samples of processed neural recordings). The optimal bandwidth of the radial basis kernel for covariance estimation is then found by minimizing the leave-one-out mean squared error of Nadaraya-Watson (NW) kernel regression using the validation set and the outer product of the validation residuals ($\boldsymbol{Z}_i - f(\boldsymbol{X}_i)$ for all $\{\boldsymbol{Z}_i, \boldsymbol{X}_i\}$ pairs in the validation data). Next, the network predicts the latent states corresponding to the test data observations (1,000 samples), while NW kernel regression predicts the latent state estimate covariances using the test data, validation data, and validation residuals. The final DKF-NN predictions are made by passing the network's predicted states and covariances, along with the aforementioned Kalman filter parameters $\boldsymbol{A}$ (state transition matrix), $\boldsymbol{G}$ (the process noise covariance, also written as $\boldsymbol{Q}$), and $\boldsymbol{V}_Z$ (the stationary covariance of $\boldsymbol{Z_i}$ without conditioning on any $\boldsymbol{Z}_{i-1}$).

Nadaraya-Watson Kernel Regression — Discriminative Kalman filtering via Nadaraya-Watson (NW) kernel regression functions similarly to how NW regression estimated the state covariance function $Q : \mathbb{R}^{10} \to \mathbb{S}^2$ in the case of neural network regression. First, the bandwidth of the radial basis kernel is optimized to minimize the leave-one out mean squared error in the estimated function $f : \mathbb{R}^{10} \to \mathbb{R}^2$ using the (3,500) training samples. Next, $f(\cdot)$ predicts the latent states (velocities) of the (1,5000) validation samples. $Q(\cdot)$ is then estimated from the resulting validation residuals in the same fashion as in neural network regression. For the (1,000) test sample observations, $f(\cdot)$ and $Q(\cdot)$ then predict the latent states and state covariances, which are processed into the final DKF-NW estimates in the same manner as in the neural network regression case.

Gaussian Process Regression — The final filtering method dependent on the Discriminative Kalman Filter involves estimating $f : \mathbb{R}^{10} \to \mathbb{R}^2$ via Gaussian process (GP) regression [20]. As with neural network and Nadaraya-Watson (NW) regression, the residuals of the estimated function $f(\cdot)$ are calculated on the validation data and then used to estimate $Q(\cdot)$. Afterwards, $f(\cdot)$ and $Q(\cdot)$ predict the latent states and covariances of the test data. The predictions undergo the DKF algorithm to produce a final DKF-GP estimate in the same manner as earlier DKF regressions.

The authors utilized the GPML MATLAB package to train their Gaussian process models [20, 21]. However, due to the lack of modern Python ports of GPML, we instead used the GaussianProcessRegressor (GPR) class from scikit-learn[5]. While the GPR class and GPML package both use algorithms from the same work [20], GPR regression is less

[5] sklearn.gaussian_process.GaussianProcessRegressor

customizable and flexible than GPML regression, resulting in worse accuracy (but improved runtime) compared to Burkhart et al. (2020) [9], despite our usage of the same kernel type (radial basis function / squared exponential). Unlike the authors, who use two separate $\mathbb{R}^{10} \rightarrow \mathbb{R}$ Gaussian process regressions to compose $f : \mathbb{R}^{10} \rightarrow \mathbb{R}^2$, we use a single $\mathbb{R}^{10} \rightarrow \mathbb{R}^2$ Gaussian process regression, since we did not observe any improvements in performance from using the former method (likely due in part to our usage of an isotropic kernel). We also did not find any improvements from using an anisotropic kernel.

Long Short-Term Memory Regression — Unlike all other regression methods, Long Short-Term Memory (LSTM) regression [22] does not use any explicit Kalman filtering framework and does not learn $f(\cdot)$ in the same manner as described in Equation 1. Instead, we and Burkhart et al. (2020) [9] used an LSTM recurrent network that learns $f(\cdot)$ conditioned on $\boldsymbol{X}_i$ and its previous two observations $\boldsymbol{X}_{i-1}$ and $\boldsymbol{X}_{i-2}$.

We were successfully able to replicate the authors' LSTM network, since they also wrote theirs in Python. However, because we used modern PyTorch while they used the older TensorFlow V1 framework, we still had to translate their architectural methods to our newer framework. In accordance with the authors' work, our model consisted of one LSTM layer of hidden dimensionality 20 that recurrently processes 3 observations from $\mathbb{R}^{10}$ before its hidden state undergoes a linear projection onto $\mathbb{R}^{20}$. Unlike the authors, we used Adam optimization (with a learning rate of 10^{-3} and an $l2$ weight penalty of 10^{-4}) and no dropout, since the LSTM performed better with those adjustments made. Note that the LSTM had significantly fewer parameters than data points, which is likely why dropout did not improve performance.

Unlike the DKF, EKF, and UKF methods, LSTM regression partitions the training and validation differently. Recall that there are 5,000 observations in the training and validation data. The LSTM training and validation data are not drawn randomly; i.e., the first 3,500 observations are for training, while the last 1,500 are for validation.

As in all previous methods, $Q : \mathbb{R}^{10} \rightarrow \mathbb{S}^2$ is learned via Nadaraya-Watson kernel regression. Interestingly, Burkhart et al. (2020) did not apply the DKF to process LSTM-estimated states. Upon further investigation, we found that their choice made empirical sense, since DKF processing worsened LSTM performance, as can be seen in Tables 4 and 5. In order to further investigate the interactions between sequence models and DKF methods, we also tried using a Transformer [23] model to estimate $f(\boldsymbol{X}_i, \boldsymbol{X}_{i-1}, \boldsymbol{X}_{i-2})$. However, given our small input timestep size of 3, we found that the LSTM architecture had superior performance in both the normalized root mean square error (nRMSE) and mean absolute angle error (MAAE) metrics with which we valuate filtering methods, required at least 10x fewer parameters, and was easier to train.

Extended and Unscented Kalman Filters — Burkhart et al. (2020) [9] used the existing native MATLAB implementations of the Extended and Unscented Kalman filters, while we used the FilterPy library [24]. Usage of the Extended Kalman Filter (EKF) [12] and Unscented Kalman Filter (EKF) [13] begins with learning the state-to-observation function $f^{-1} : \mathbb{R}^2 \rightarrow \mathbb{R}^{10}$, unlike the previous DKF methods. To learn $f^{-1}(\cdot)$, we used a neural network with the same architecture (albeit with flipped input and output dimensionalities) as that of the DKF-NN method (2 hidden layers of 10 hyperbolic tangent neurons), since the authors' original $f^{-1}(\cdot)$ network faced the same issues of reproduction in Python as their $f(\cdot)$ network. We also used the same optimization method, hyperparameters, and training data as in the DKF-NN method (4000 iterations, 10^{-3} learning rate, and a 10^{-4} $l2$ weight penalty). For both the EKF and the UKF, the observation noise $\boldsymbol{R}$ is estimated as the covariance of the residuals evaluated over the validation data (i.e., the covariance

of $\boldsymbol{X}_i - f(\boldsymbol{Z}_i)$ for all $\{\boldsymbol{X}_i, \boldsymbol{Z}_i\}$ pairs in the validation data).

The EKF uses the learned $f^{-1}(\cdot)$ as its observation function. The Jacobian of the function is available through PyTorch, and is supplied to the EKF algorithm. Along with the aforementioned information from $f^{-1}(\cdot)$, the EKF uses the residual-estimated $\boldsymbol{R}$ and the Kalman filter parameters $\boldsymbol{A}$ (the state transition matrix), $\boldsymbol{G}$ (the process noise, also written as $\boldsymbol{Q}$), and an initial covariance estimate $\boldsymbol{P}_0 = \boldsymbol{V}_Z$ to iteratively calculate predictions over the test data.

The UKF uses the same observation function $f^{-1}(\cdot)$ as the EKF. While the UKF can operate with explicitly nonlinear state transitions $F : \mathbb{R}^2 \to \mathbb{R}^2$ (unlike the EKF), the state transition function used here (in accordance with the authors' work) is set as multiplication by the Kalman state transition matrix: $F(\boldsymbol{Z_i}) = \boldsymbol{A}\boldsymbol{Z}_i$. The UKF uses the same Kalman parameters as the EKF ($\boldsymbol{G}$, $\boldsymbol{R}$, and $\boldsymbol{P}_0 = \boldsymbol{V}_Z$) and generates predictions over the test data in the same fashion as the EKF. One key difference between the UKF algorithm here and the UKF algorithm used by Burkhart et al. (2020) [9] is that MATLAB differs from FilterPy in how the number of sample (sigma) points are calculated.

4 Results

	Trial 1	Trial 2	Trial 3	Trial 4	Trial 5	Trial 6	Average
Kalman	0.765 | **0.765**	0.945 | **0.942**	0.788 | **0.788**	0.792 | **0.793**	0.779 | **0.780**	0.761 | **0.765**	0.805 | **0.805**
DKF-NW	–19% | **–21%**	–18% | **–18%**	–9% | **–17%**	–21% | **–23%**	–19% | **–20%**	–20% | **–23%**	–18% | **–20%**
DKF-GP	–7% | **–21%**	–11% | **–19%**	–8% | **–15%**	–9% | **–20%**	–12% | **–18%**	–9% | **–20%**	–9% | **–19%**
DKF-NN	–23% | **–19%**	0% | **–15%**	–7% | **–13%**	–15% | **–13%**	–19% | **–13%**	–22% | **–17%**	–14% | **–15%**
LSTM	–22% | **–15%**	–1% | **–19%**	–21% | **–16%**	–19% | **–13%**	–21% | **–16%**	–25% | **–11%**	–18% | **–15%**
EKF	–1% | **2%**	7% | **24%**	8% | **12%**	18% | **18%**	11% | **12%**	6% | **3%**	8% | **12%**
UKF	1% | **2%**	1% | **31%**	6% | **18%**	11% | **18%**	5% | **15%**	3% | **6%**	4% | **15%**

Table 2. Normalized RMSE (nRMSE) between the predicted and true test velocities using one random seed, as in Table 1 from Burkhart et al. (2020) [9]. Bolded values indicate the authors' published results, while highlighted values denote our best result for each trial. Note that predicting identically zero would yield a nRMSE of 1.

	Trial 1	Trial 2	Trial 3	Trial 4	Trial 5	Trial 6	Average
Kalman	0.884 | **0.889**	0.957 | **0.955**	1.026 | **1.025**	0.930 | **0.933**	0.966 | **0.964**	0.926 | **0.926**	0.948 | **0.949**
DKF-NW	–14% | **–15%**	0% | **–1%**	–21% | **–20%**	–15% | **–17%**	–25% | **–25%**	–29% | **–28%**	–17% | **–18%**
DKF-GP	–5% | **–11%**	10% | **7%**	–19% | **–22%**	–7% | **–16%**	–17% | **–24%**	–21% | **–25%**	–10% | **–15%**
DKF-NN	–9% | **–7%**	11% | **–2%**	–15% | **–17%**	–11% | **–16%**	–19% | **–21%**	–21% | **–23%**	–11% | **–14%**
LSTM	–6% | **–2%**	13% | **–2%**	–18% | **–12%**	–12% | **–6%**	–20% | **–10%**	–20% | **–8%**	–11% | **–7%**
EKF	1% | **4%**	12% | **3%**	2% | **–2%**	5% | **–4%**	–8% | **–8%**	–1% | **–7%**	2% | **–2%**
UKF	–1% | **0%**	9% | **3%**	1% | **–3%**	2% | **–3%**	–10% | **–8%**	–6% | **–6%**	–1% | **–3%**

Table 3. Mean Absolute Angle Error (MAAE, in radians) of the predicted and true test velocities using one random seed, as in Table 2 from Burkhart et al. (2020) [9]. Bolded values indicate the authors' published results, while highlighted values denote our best result for each trial. Note that chance prediction would yield a MAAE of $\pi/2 \approx 1.57$ radians. The authors argued that MAAE may be a more salient metric for neuroprosthetic applications.

After reproducing the regression methods implemented by Burkhart et al. (2020) [9], we evaluated their performance on the held-out 1,000 samples of test data of the first six of twelve trials, in the same manner as the original authors and with the same random seed. Specifically, we calculated the normalized root-mean square error (nRMSE) and mean absolute angle error (MAAE) between the predicted and ground truth test data velocities, as shown in Tables 2 and 3, respectively. Bolded values indicate the authors'

original results. While we were unable to replicate most of the exact numerical results from Burkhart et al. (2020) [9], we faithfully reproduced the trends in performance from the authors' work. Notable exceptions include worsened DKF-GP performance (due to the difficulty of translating GPML computations to Python) and heightened LSTM performance (due to added hyperparameter tuning). Although the DKF-NW and LSTM regression methods had the same average performance over the six trials, our results agreed with those of Burkhart et al. (2020) [9] in that the DKF-NW triumphed as the best regression method due to its superior performance with respect to the MAAE metric.

	Trial 1	Trial 2	Trial 3	Trial 4	Trial 5	Trial 6	Trial 9	Trial 10	Average
K	0.76	0.94	0.79	0.79	0.78	0.76	1.01	0.92	0.84
NW	-15%	-18%	-18%	-24%	-20%	-21%	-5%	3%	-15%
DKF-NW	-18%	-18%	-12%	-19%	-18%	-20%	-5%	2%	-14%
GP	-3%	-9%	-7%	-8%	-10%	-6%	-7%	-14%	-8%
DKF-GP	-6%	-11%	-8%	-8%	-12%	-8%	-8%	-16%	-10%
NN	-19%	-17%	-21%	-23%	-20%	-23%	-5%	-7%	-17%
DKF-NN	-20%	-11%	-12%	-16%	-17%	-20%	-4%	-9%	-14%
LSTM	-22%	12%	-23%	-21%	-23%	-25%	-7%	-12%	-15%
DKF-LSTM	-15%	84%	-8%	-7%	-15%	-14%	-4%	-12%	1%
EKF	2%	10%	11%	18%	14%	8%	5%	12%	10%
UKF	-1%	-1%	6%	15%	10%	4%	-5%	2%	4%

Table 4. Average Normalized RMSE (nRMSE) between the predicted and true test velocities, with and without DKF filtering, evaluated over ten different random seeds. The best results from each trial are highlighted. Trials 7, 8, 11, and 12 did not have the requisite number of samples and were thus not included.

	Trial 1	Trial 2	Trial 3	Trial 4	Trial 5	Trial 6	Trial 9	Trial 10	Average
K	0.88	0.96	1.03	0.93	0.97	0.93	0.97	1.03	0.96
NW	-8%	1%	-21%	-15%	-21%	-25%	-1%	-6%	-12%
DKF-NW	-14%	-1%	-21%	-16%	-25%	-28%	-5%	-5%	-14%
GP	2%	12%	-18%	-3%	-13%	-16%	6%	-4%	-4%
DKF-GP	-5%	11%	-19%	-7%	-19%	-20%	0%	-8%	-8%
NN	1%	3%	-17%	-7%	-15%	-15%	4%	3%	-5%
DKF-NN	-6%	2%	-17%	-10%	-19%	-20%	-3%	1%	-9%
LSTM	-6%	15%	-20%	-13%	-20%	-20%	-3%	-5%	-9%
DKF-LSTM	-6%	24%	-18%	-12%	-20%	-20%	-2%	-5%	-7%
EKF	0%	8%	1%	1%	-4%	-2%	3%	19%	3%
UKF	-4%	6%	0%	0%	-8%	-7%	-1%	13%	0%

Table 5. Average Mean Absolute Angle Error (MAAE) between the predicted and true test velocities, with and without DKF filtering, evaluated over ten different random seeds, as in Table 4. The best results from each trial are highlighted.

To bring further insight into the performance of the Discriminative Kalman filter in neuroscientific contexts, we evaluated the aforementioned regression methods with and without DKF filtering applied. Specifically, we measured their average performance (see Tables 4 and 5) on all eight of the twelve trials that had the requisite number of samples (at least 6,000) using ten different random seeds. To our surprise, we found that unfiltered neural network regression best minimized nRMSE, but that the DKF-NW had the lowest MAAE. Interestingly, DKF application for neural networks increases nRMSE while decreasing MAAE.

5 Conclusion

This partial replication study confirms the most salient results from Burkhart et al. (2020) [9] concerning the performance of their Discriminative Kalman filter (DKF) on publicly available neuroprosthetic data. We not only successfully affirmed that the DKF-NW had the best overall performance, but we also conducted further tests that prove the efficacy of the DKF while showing the differences that may occur between different metrics for evaluating filter performance.

While DKF methods improved over the Kalman baselines, the closeness in performance of Kalman and DKF methods to the trivial baseline of predicting identically zero, especially when using the nRMSE metric, indicates that future work is needed to improve such filters. For example, future endeavors could include Discriminative Kalman Filter incorporation into more modern approaches in neural latent state estimation, such as sequential/dynamical autoencoders, modern state-space models, Kalman networks, or transformers [6, 25, 26]. While we did not find our transformer architecture outperformed LSTMs, it is certainly possible that such an architecture could surpass LSTM performance, especially in trials of much longer durations with longer-range dependencies. Additionally, the effects of DKF application on neural network or LSTM regression are not entirely clear, and could stand to be elucidated in future works.

6 Acknowledgements

We give thanks to Flint et al. (2012) [15] and Walker and Kording (2013) [16] for collecting and hosting the primate reach data, Burkhart et al. (2020) [9] for their mathematical insights into Bayesian filtering, and Caleb Kemere and Richard G. Baraniuk for discussion of this endeavor.

References

1. Y. S. Sonam. "A Review Paper on Brain Computer Interface." In: **International Journal of Engineering Research & Technology NCETEMS** 3.10 (2015). DOI: 10.17577/IJERTCONV3IS10102.
2. F. R. Willett, D. T. Avansino, L. R. Hochberg, J. M. Henderson, and K. V. Shenoy. "High-performance brain-to-text communication via handwriting." In: **Nature** 593.7858 (2021).
3. K. A. Yildiz, A. Y. Shin, and K. R. Kaufman. "Interfaces with the peripheral nervous system for the control of a neuroprosthetic limb: a review." In: **Journal of neuroengineering and rehabilitation** 17.1 (2020).
4. S. Saha, K. A. Mamun, K. Ahmed, R. Mostafa, G. R. Naik, S. Darvishi, A. H. Khandoker, and M. Baumert. "Progress in brain computer interface: Challenges and opportunities." In: **Frontiers in Systems Neuroscience** 15 (2021).
5. R. E. Kalman. "A New Approach to Linear Filtering and Prediction Problems." In: **Journal of Basic Engineering** 82.1 (1960). DOI: 10.1115/1.3662552.
6. C. Pandarinath, D. J. O'Shea, J. Collins, R. Jozefowicz, S. D. Stavisky, J. C. Kao, E. M. Trautmann, M. T. Kaufman, S. I. Ryu, L. R. Hochberg, et al. "Inferring single-trial neural population dynamics using sequential auto-encoders." In: **Nature methods** 15.10 (2018).
7. A. K. Vaskov, Z. T. Irwin, S. R. Nason, P. P. Vu, C. S. Nu, A. J. Bullard, M. Hill, N. North, P. G. Patil, and C. A. Chestek. "Cortical decoding of individual finger group motions using ReFIT Kalman filter." In: **Frontiers in neuroscience** 12 (2018).
8. N. Mudrik, Y. Chen, E. Yezerets, C. J. Rozell, and A. S. Charles. "Decomposed Linear Dynamical Systems (dLDS) for learning the latent components of neural dynamics." In: **arXiv preprint arXiv:2206.02972** (2022). DOI: 10.48550/ARXIV.2206.02972.
9. M. C. Burkhart, D. M. Brandman, B. Franco, L. R. Hochberg, and M. T. Harrison. "The Discriminative Kalman Filter for Bayesian Filtering with Nonlinear and Nongaussian Observation Models." In: **Neural Computation** 32.5 (2020). DOI: 10.1162/neco_a_01275.
10. T. M. McKenna, T. A. McMullen, and M. F. Shlesinger. "The brain as a dynamic physical system." In: **Neuroscience** 60.3 (1994).

11. L. Rabiner and B. Juang. "An introduction to hidden Markov models." In: **IEEE ASSP Magazine** 3.1 (1986). DOI: 10.1109/MASSP.1986.1165342.
12. M. I. Ribeiro. "Kalman and extended kalman filters: Concept, derivation and properties." In: **Institute for Systems and Robotics** 43 (2004).
13. E. A. Wan and R. Van Der Merwe. "The unscented Kalman filter for nonlinear estimation." In: **Proceedings of the IEEE 2000 Adaptive Systems for Signal Processing, Communications, and Control Symposium**. 2000.
14. M. C. Burkhart, D. M. Brandman, C. E. Vargas-Irwin, and M. T. Harrison. "The discriminative Kalman filter for nonlinear and non-Gaussian sequential Bayesian filtering." In: **arXiv preprint arXiv:1608.06622** (2016). DOI: 10.48550/ARXIV.1608.06622.
15. R. D. Flint, E. W. Lindberg, L. R. Jordan, L. E. Miller, and M. W. Slutzky. "Accurate decoding of reaching movements from field potentials in the absence of spikes." In: **Journal of Neural Engineering** 9.4 (2012). DOI: 10.1088/1741-2560/9/4/046006.
16. B. Walker and K. Kording. "The Database for Reaching Experiments and Models." In: **PLOS ONE** 8.11 (2013). DOI: 10.1371/journal.pone.0078747.
17. E. A. Nadaraya. "On estimating regression." In: **Theory of Probability & Its Applications** 9.1 (1964).
18. F. D. Foresee and M. T. Hagan. "Gauss-Newton approximation to Bayesian learning." In: **Proceedings of international conference on neural networks (ICNN'97)**. Vol. 3. 1997.
19. S. Ruder. "An overview of gradient descent optimization algorithms." In: **arXiv preprint arXiv:1609.04747** (2016). DOI: 10.48550/ARXIV.1609.04747.
20. C. E. Rasmussen and C. K. I. Williams. **Gaussian Processes for Machine Learning**. 2005. DOI: 10.7551/mitpress/3206.001.0001.
21. C. E. Rasmussen and H. Nickisch. "Gaussian processes for machine learning (GPML) toolbox." In: **The Journal of Machine Learning Research** 11 (2010).
22. S. Hochreiter and J. Schmidhuber. "Long short-term memory." In: **Neural computation** 9.8 (1997).
23. A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, Ł. Kaiser, and I. Polosukhin. "Attention is all you need." In: **Advances in neural information processing systems** 30 (2017).
24. R. Labbe. **Kalman and Bayesian filters in Python**. 2015.
25. L. Girin, S. Leglaive, X. Bie, J. Diard, T. Hueber, and X. Alameda-Pineda. "Dynamical Variational Autoencoders: A Comprehensive Review." In: **Foundations and Trends® in Machine Learning** 15.1-2 (2021). DOI: 10.1561/2200000089.
26. R. Liu, M. Azabou, M. Dabagia, J. Xiao, and E. L. Dyer. "Seeing the forest and the tree: Building representations of both individual and collective dynamics with transformers." In: **Advances in neural information processing systems** (2022). DOI: 10.48550/ARXIV.2206.06131.