Applying Machine Learning Algorithms for Neural Decoding

Vilde Ung
(Dated: December 17, 2022)

In this study, we compare the performance of feedforward neural networks, support vector regression, and linear regression for neural decoding. The methods are applied to a data set of 164 spike trains recorded from the primary motor cortex to predict movement velocity. We find that the feedforward neural network outperforms the other methods, explaining 89.2% of the variance in the data. By doing a thorough search for optimal hyperparameters, we were able to improve the performance of the methods compared to that of a similar study analysing the same data set. We found that the algorithms' performance was relatively robust across different hyperparameters, but that particularly regularisation improved performance of both the linear and nonlinear methods. We also discuss some of the challenges that arise in relation to feature engineering and model interpretability, which is relevant to the goal of better understanding the relationship between stimuli and neural activity. GitHub repository: https://github.com/ungvilde/FYS-STK4155/tree/main/project3

I. INTRODUCTION

Neural decoding aims to reconstruct information about the external world from the recorded activity of neurons. Neurons transmit information using rapid electrical impulses, called action potentials, and it is believed that the timing of action potentials encode biologically relevant information. Decoding neural activity is highly relevant both in engineering and neuroscience research. For example, brain-computer interfaces attempt to use recorded activity in the brain to control outputs in real time, such as controlling the movement of a robotic arm [1]. Additionally, by successfully decoding neural activity we might gain deeper insight into how the brain represents and processes information [2].

Essentially, a decoder approximates the functional relationship between brain activity and output variables. In many cases, decoders that assume a linear relationship have proved somewhat successful, but with more advanced machine learning algorithms there is the potential to greatly improve performance and to better understand the neural code [3]. In particular, deep learning algorithms, like feedforward neural networks and recurrent networks, have proved to be useful for improving the accuracy and flexibility of decoders [4, 5].

In this study, we look at two more sophisticated machine learning algorithms, namely feedforward neural networks and support vector regression, and compare their performance against a linear model. We train the learning algorithms on real data, recorded from a population of neurons in the primary motor cortex of a macaque monkey, to predict the velocity of a cursor on a screen that the monkey controlled. We do grid searches to find the optimal hyperparameters of each algorithm, and compare the results against a similar analysis done on the same data set [6].

The paper is structured as follows: In Section II we outline the learning algorithms and the relevant theory; in Section III we present our results; in Section IV we discuss our findings and their implications; and finally, in Section V we conclude the study and summarise our analysis.

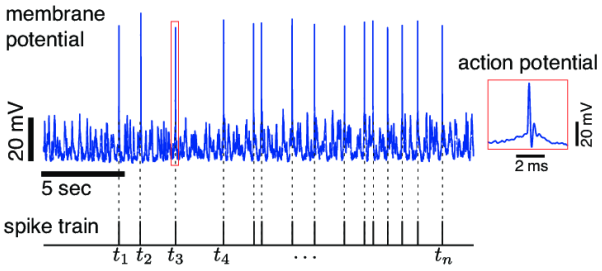

Figure 1. The membrane potential of a single neuron recorded in continuous time (the blue trace), and the spike train of that neuron, represented by a sequence of spike times $(t_1, t_2, t_3, t_4, \dots t_n)$ where action potentials were detected. The shape of a single action potential is presented on the right-hand side. The figure is taken from [8].

II. METHODS

A. Spike Train Data

In this study, we will apply machine learning methods to *spike train* data, which is the recorded activity of individual neurons over time. When a neuron is activated, the electric potential across the cell membrane changes rapidly, and we say that the neuron "spikes" or "fires". This rapid change in membrane potential is called an *action potential*, and it is thought to be the base unit of information in the brain, conveying information through its timing [7]. In particular, we are interested in the rate with which action potentials occur over some time interval Δt, also called the *firing rate* (FR):

$$\text{FR} = \frac{\text{Number of spikes}}{\Delta t}.$$

We usually consider all action potentials to be uniform in terms of strength and duration ($1 - 2$µs), and so a sequence of action potentials is fully characterised by the times at which they occurred.

By inserting electrodes into individual neurons, we can record the spikes of neurons over time as a binary time

series, where we record "1" whenever the membrane potential passes a threshold value and "0" otherwise. In Figure 1 the relationship between the membrane potential and the recorded spike times is visualised.

Here, we will look at spike trains recorded from 164 neurons in the primary motor cortex (M1) of a macaque monkey. The monkey performed a motor task that involved reaching a series of targets using a manipulandum that controlled a cursor on a screen. The movement velocity in the x- and y-direction of the cursor was recorded, and will be applied as the target variable for decoding in this study. See [9] for further details, and see `https://github.com/kordinglab/neural_decoding` to access the data set and pertinent code.

B. Neural Decoding

With neural decoding, we aim to make predictions about some target variable based on the recorded activity of neurons, which is essentially a regression problem [6]. The name implies that the electrical activity in neurons represents, or encodes, information about the external world. It has been shown, for example, that specific regions of neurons in the visual cortex become active in response to seeing lines at specific angles [10]. Neural decoding is particularly relevant in brain-computer interfaces, where the goal is to use brain signals to control some output variable, like movement, in real time.

In relating neural activity to the target variable, there are two main stages [4]. The first is to preprocess the data, constructing relevant features from the raw data of recorded activity that can be applied to the learning algorithm. This will typically involve modelling the firing rate, where we can account for time-delayed firing rate effects as well. The second is to construct some mapping to make predictions, whereby we transform the relevant features to the target representation. This is where we implement a learning algorithm.

C. Data Preprocessing

Due to the fact that spike trains are recorded in continuous time, we need to determine the temporal resolution for making predictions. The data is recorded over a time interval $(0, T]$. We discretise the data, so the time interval is divided into time bins of size Δt. This gives us a total number of $K^* = T/\Delta t$ time bins. We then count the number of recorded spikes within each time bin, effectively estimating the firing rates of the neurons. Similarly, the outcome data is discretised and averaged at each time bin.

With spike train data, we want to choose some time period of recorded activity with which to make predictions. The recorded spikes within this time period will be the features of our model. The simplest way to do this, is by using the spikes that are concurrent with the outcome, but studies have shown that including a time-delay can make the predictions more robust [11]. This makes sense physically, as the physiological relationship between neurons firing and some behaviour being executed introduces a time-delay between events. For example, we might want to use the recorded spiking activity from 500 µs before and up to the outcome occurred, to make a prediction. In Figure 2 we have visualised how spike trains of N neurons are discretised into time bins, and how the activity from B previous time bins is used to predict the output.

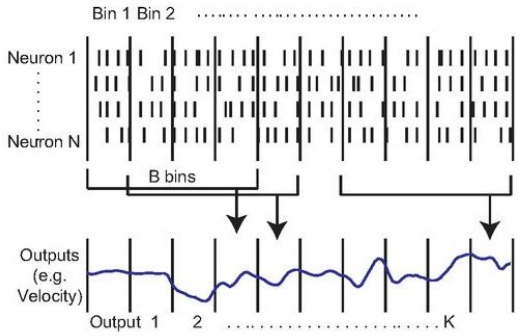

Figure 2. The spike trains of N neurons are divided into time bins, and the spike counts from B time bins are used to predict the outcome. The figure is taken from [9].

In this study, we use B time bins from each of the N recorded neurons to predict the output. Specifically, we use the $B-1$ previous bins as well as the concurrent bin.

Note that when $B > 1$, we will not predict the first $B-1$ outcomes, because they do not have the necessary feature values. This gives us a total number of $N \cdot B$ features, so the design matrix $\mathbf{X}$ will have $N \cdot B$ columns and K rows, where $K = K^* - B + 1$ is the number of targets to predict. The target matrix $\mathbf{Y}$ will have K rows and d columns, where d is the dimension of the target to predict. Specifically, we will predict velocities in both the x- and y-directions, so $d = 2$.

D. The Linear Model

With the linear model, also referred to as a *Wiener filter* in the neural decoding literature, the goal is to find a linear relationship between the features and the target variables. Despite their simplicity, linear decoders have been shown to be perform reasonably well, specially on noisy data sets [9]. Also, linear decoders have previously been successful at making predictions related to motor action, such as in [12, 13].

We will perform multivariate linear regression to predict the velocity in the x- and y-direction, which is equivalent to fitting two linear regression models. We first outline the model for the velocity in the x-direction, which is analogous to the model for the velocity in the y-direction.

The basic form of the model is

$$y_k = \beta_0 + \sum_{b=0}^{B-1} \mathbf{x}_{k-b}^T \boldsymbol{\beta}_b + \epsilon_k.$$

Here, we have the target variable y_k, which is the average velocity in time bin k, and an intercept β_0. The vector $\boldsymbol{\beta}_b$ holds the weights for the spikes with time-delay b. Furthermore, $\mathbf{x}_{k-b}$ holds the spikes of the N recorded neurons in time bin $k-b$. Finally, $\epsilon_k \sim \mathcal{N}(0, \sigma^2)$ is a normal noise term.

In matrix form, the model is given as

$$\mathbf{y} = \mathbf{X}\boldsymbol{\beta} + \boldsymbol{\epsilon},$$

where $\mathbf{y}$ is a vector of length K holding the target variables. We have the $K \times p$ design matrix $\mathbf{X}$, which holds the spike count for all the N recorded neurons over the B different time lags, as well as an intercept column. Furthermore, $\boldsymbol{\beta}$ is a vector of length p holding the model parameters, and $\boldsymbol{\epsilon}$ is a vector of length K with normally distributed noise terms. Here $p = N \cdot B + 1$ includes the number of feature columns and the intercept column.

To find the optimal parameters, we will apply ordinary least-squares methods, which involves minimising the mean squared-error cost function

$$C(\boldsymbol{\beta}) = ||\mathbf{X}\boldsymbol{\beta} - \mathbf{y}||_2^2 + \lambda ||\boldsymbol{\beta}||_2^2,$$

where $||\cdot||_2$ is the $\mathcal{L}^2$-norm. We incorporate a regularisation term λ to control for overfitting. Solving $\frac{\partial C(\boldsymbol{\beta})}{\partial \boldsymbol{\beta}} = 0$ shows that we can find the optimal parameters as

$$\hat{\boldsymbol{\beta}} = (\mathbf{X}^T\mathbf{X} + \lambda \mathbf{I})^{-1}\mathbf{X}^T\mathbf{y}.$$

Note that the model assumes that the linear relationship between the spiking and the velocities is constant in time.

E. Feedforward Neural Networks

Deep neural networks allow for flexible, nonlinear decoders, and are the most frequently used in neuroscience [4]. Feedforward neural networks (FFNNs) combine linear matrix multiplications and nonlinear operations, like the sigmoid function or the rectifier, with a layered network architecture of fully connected nodes. The layers are organised in a chain structure, where the nodes of each layer is computed from those of the previous layer.

The first layer uses the design matrix $\mathbf{X}$ with $p = N \cdot B$ feature columns and K rows as its input, and is given by

$$\mathbf{h}^{(1)} = g\left(\mathbf{W}^{(1)T}\mathbf{X} + \mathbf{b}^{(1)}\right).$$

Any following layers are computed as

$$\mathbf{h}^{(\ell)} = g\left(\mathbf{W}^{(\ell)T}\mathbf{h}^{(\ell-1)} + \mathbf{b}^{(\ell)}\right).$$

Finally we get the output layer as

$$\hat{\mathbf{Y}} = \mathbf{W}^{(L)T}\mathbf{h}^{(L-1)} + \mathbf{b}^{(L)},$$

where L is the total number of layers in the network, also referred to as the *depth* of the network. The weights and biases of layer ℓ are denoted by $\mathbf{W}^{(\ell)}$ and $\mathbf{b}^{(\ell)}$, respectively. These are the model parameters. We apply a nonlinear *activation function* at each layer, denoted by g, which is the same for all layers, except in the final layer. The output activation function is simply a linear function. From this we see that the neural network is essentially a composition of L functions, of the form

$$\hat{\mathbf{Y}} = f(\mathbf{X}; \boldsymbol{\theta}) = f^{(L)} \circ f^{(L-1)} \circ \cdots \circ f^{(1)}(\mathbf{X}),$$

where $\hat{\mathbf{Y}}$ is a $K \times d$ matrix of the predicted values, and $f^{(\ell)}$ denotes the functional transformation applied in layer ℓ. Here, $\boldsymbol{\theta}$ represents the parameters of the model, i.e. the weights and biases of the network.

In contrast to the linear model, the FFNN does not constrain the functional relationship between $\mathbf{X}$ and $\mathbf{Y}$ to any particular form, allowing for greater flexibility. This usually requires more model parameters, but it can be an advantage when modelling large, complex data sets. Additionally, we note that the same FFNN predicts the velocities in the x- and y-direction simultaneously. With the linear model, these are predicted independently.

With a linear output activation, we can fit the FFNN by minimising the mean squared error,

$$C(\boldsymbol{\theta}) = \sum_{i=1}^{d} ||f_i(\mathbf{X}; \boldsymbol{\theta}) - \mathbf{y}_i||_2^2 + \lambda ||\boldsymbol{\theta}||_2^2.$$

Here, λ is the regularisation parameter. Note that we sum over the d different outputs $\mathbf{y}_i$ and $f_i(\mathbf{X}; \boldsymbol{\theta})$, which represent the columns in the target matrix $\mathbf{Y}$ and prediction matrix $\hat{\mathbf{Y}}$, respectively.

The minimisation is done by first using the backpropogation algorithm to compute the gradient $\nabla_{\boldsymbol{\theta}} C(\boldsymbol{\theta})$, and then using stochatic gradient descent for finding the optimal parameters $\hat{\boldsymbol{\theta}}$. These algorithms are explained in depth in previous work, see [14] for details.

F. Support Vector Regression

Let $\mathbf{x}_i$ be a feature vector of p data points with a corresponding target value y_i, for $i = 1, 2, \ldots, K$. The main goal of support vector regression (SVR) is to find a function $f(\mathbf{x})$ that has at most $\epsilon > 0$ deviation from the target values, while being as "flat" as possible.

We first introduce the model in terms of a linear function,

$$f(\mathbf{x}) = \mathbf{x}^T\boldsymbol{\beta} + \beta_0.$$

In this case, the "flatness" criterion means that we want to minimise $\frac{1}{2}||\boldsymbol{\beta}||^2$. Furthermore, we want to satisfy

$$|f(\mathbf{x}_i) - y_i| \leq \epsilon. \tag{1}$$

However, there might not exist a function f that satisfies (1), so we want to allow for some errors. We therefore introduce *slack variables*, denoted by ξ_i and ξ_i^*, and update the constraints of our problem:

$$\begin{aligned} &\text{minimise} \quad \frac{1}{2}||\boldsymbol{\beta}||^2 + C\sum_{i=1}^{N}(\xi_i + \xi_1^*), \\ &\text{while satisfying} \quad \begin{cases} f(\mathbf{x}_i) - y_i \leq \epsilon + \xi_i \\ y_i - f(\mathbf{x}_i) \leq \epsilon + \xi_i^* \\ \xi_i, \xi_i^* \geq 0. \end{cases} \end{aligned} \tag{2}$$

The $C > 0$ is included to parameterise the trade-off between the flatness of f and how much the model deviates from the target values by more than ϵ. The smaller C is, the more regularised the model is. Note that deviations greater than ϵ have a linear cost, while deviations smaller than ϵ have no cost.

We can perform the minimisation under the given constraints in (2) by introducing Lagrange multipliers $\alpha_i \geq 0$, $\alpha_i^* \geq 0$, $\eta_i \geq 0$ and $\eta_i^* \geq 0$, and optimising the Lagrangian function

$$\begin{aligned} \mathcal{L} = &\frac{1}{2}||\boldsymbol{\beta}||^2 + C\sum_{i=1}^{K}(\xi_i + \xi_1^*) - \sum_{i=1}^{K}(\eta_i\xi_i + \eta_i^*\xi_i^*) \\ &- \sum_{i=1}^{K}\alpha_i(\epsilon + \xi_i - y_i + \mathbf{x}^T\boldsymbol{\beta} + \beta_0) \\ &- \sum_{i=1}^{K}\alpha_i^*(\epsilon + \xi_i^* + y_i - \mathbf{x}^T\boldsymbol{\beta} - \beta_0). \end{aligned}$$

By setting the partial derivatives of $\mathcal{L}$ with respect to the primary parameters $(\boldsymbol{\beta}, \beta_0, \xi_i, \xi_i^*)$ to zero, and doing the necessary substitutions, it can be shown that

$$\begin{aligned} \boldsymbol{\beta} &= \sum_{i=1}^{K}(\alpha_i - \alpha_i^*)\mathbf{x}_i \\ f(\mathbf{x}_i) &= \sum_{i=1}^{K}(\alpha_i - \alpha_i^*)\langle \mathbf{x}_i, \mathbf{x}_i\rangle + \beta_0. \end{aligned} \tag{3}$$

Here, $\langle\cdot,\cdot\rangle$ is the dot product in $\mathbb{R}^p$. The Lagrange multipliers α_i and α_i^* are given by

$$\begin{aligned} &\text{maximising} \quad -\frac{1}{2}\sum_{i,j=1}^{K}(\alpha_i - \alpha_i^*)(\alpha_j - \alpha_j^*)\langle \mathbf{x}_i, \mathbf{x}_j\rangle \\ &\qquad\qquad -\epsilon\sum_{i=1}^{K}(\alpha_i + \alpha_i^*) + \sum_{i=1}^{K} y_i(\alpha_i - \alpha_i^*), \\ &\text{while satisfying} \quad \begin{cases} \alpha_i, \alpha_i^* \in [0, C] \\ \sum_{i=1}^{K}(\alpha_i - \alpha_i^*) = 0. \end{cases} \end{aligned}$$

See [15, 16] for further details on this derivation, as well as for details on how to compute β_0. Note that the model as given in (3) only depends on the dot product of the data vectors $\mathbf{x}_i$, meaning we do not need to compute $\boldsymbol{\beta}$ directly.

1. Nonlinear Kernels

We can make the model nonlinear by introducing a nonlinear map $\boldsymbol{\Phi} : \mathbb{R}^p \to \mathcal{F}$, where $\mathcal{F}$ is the *feature space*. Then the nonlinear model is defined as

$$f(\mathbf{x}) = \Phi(\mathbf{x})^T\boldsymbol{\beta} + \beta_0.$$

We solve this by applying the same algorithm as with the linear model, but now in feature space.

By introducing the map Φ, we replace the dot product in our algorithm by a *kernel function*, defined as $k(\mathbf{x}, \mathbf{x}') = \langle\Phi(\mathbf{x}), \Phi(\mathbf{x}')\rangle$. In the new optimisation problem, we

$$\begin{aligned} &\text{maximise} \quad -\frac{1}{2}\sum_{i,j=1}^{K}(\alpha_i - \alpha_i^*)(\alpha_j - \alpha_j^*)k(\mathbf{x}_i, \mathbf{x}_j) \\ &\qquad\qquad -\epsilon\sum_{i=1}^{K}(\alpha_i + \alpha_i^*) + \sum_{i=1}^{K} y_i(\alpha_i - \alpha_i^*), \\ &\text{while satisfying} \quad \begin{cases} \alpha_i, \alpha_i^* \in [0, C] \\ \sum_{i=1}^{K}(\alpha_i - \alpha_i^*) = 0. \end{cases} \end{aligned}$$

This yields

$$\begin{aligned} \boldsymbol{\beta} &= \sum_{i=1}^{K}(\alpha_i - \alpha_i^*)\Phi(\mathbf{x}_i) \\ f(\mathbf{x}_i) &= \sum_{i=1}^{K}(\alpha_i - \alpha_i^*)k(\mathbf{x}_i, \mathbf{x}_i) + \beta_0. \end{aligned}$$

The strength of this approach, known as the *kernel trick*, is that we do not need to explicitly perform the mapping to feature space – we just evaluate the kernel.

The kernel function needs to be an inner product in the feature space. Common choices are the polynomial kernel or the gaussian radial basis function. In this study we applied the latter option, given by

$$k(\mathbf{x}, \mathbf{x}') = \exp\left(-\gamma||\mathbf{x} - \mathbf{x}'||^2\right).$$

The $\gamma > 0$ parameter determines how much influence a training data point has on the resulting model fit.

To compute α_i and α_i^* we use the `scikit-learn` Python library [17], and so we will not go into further detail on this. Note that the SVR model will predict the x- and y-velocities independently.

G. Scoring Metric

We evaluate the goodness of fit using the coefficient of determination R^2 , which quantifies the proportion of variance in the target variable that is explained by the features of the model. The R^2 score is given by

$$R^2 = 1 - \frac{\sum_{i=1}^{K}(y_i - \hat{y}_i)^2}{\sum_{i=1}^{K}(y_i - \bar{y}_i)^2},$$

where $\bar{y}_i = \frac{1}{K}\sum_{i=1}^{K} y_i$ is the average of the target variables. Here, y_i denotes the ground truth, i.e. the values to predict, and $\hat{y}_i$ denotes the predicted values.

Note that, in this study we predict two target variables, so the reported R^2 is to be understood as the average R^2 of the two targets.

Additionally, we report the root mean squared error (RMSE), computed as

$$\text{RMSE} = \sqrt{\text{MSE}} = \sqrt{\frac{1}{K}\sum_{i=1}^{K}(y_i - \hat{y}_i)^2}.$$

Also here, we report the average of the two target values.

III. RESULTS

We investigated the performance of FFNN, SVR, and a linear model on spike train data, doing a search to find the optimal hyperparameters for each algorithm. The goal was to find good hyperparameters, and to possibly outperform the results obtained by Glaser et al. in [6].

To make our analysis comparable to the results found by Glaser et al., the data preprocessing mirrored their choices, and the Python code used for data preprocessing is based on the code from their tutorial. In preprocessing the data, we used a time resolution of Δt =50 µs, and a time-delay of 700 µs. That is, we used the concurrent time bin and 13 preceding time bins to predict the velocity, a total of $B = 14$ bins. As the data set consists of spike trains from $N = 164$ neurons, we had $N \cdot B = 164 \cdot 14 = 2296$ features. The data set consisted of 21 minutes of recorded activity, and there was on average 6.7 spikes per second.

The data set was standardised as z-scores, by centering and scaling it to have unit variance. Note that in doing SVR we also standardised the target values. We split the data set into a training set and a test set, where the training set consisted of 70% of the full data. In searching for optimal hyperparameters we did 5-fold cross validation using the training data. The model performance was finally evaluated in the unseen test set.

The methods were in large part implemented using the `scikit-learn` Python library [17], because our own Python implementations turned out to be very slow on such a large data set, which would have limited us from doing an extensive hyperparameter search.

A. The Linear model

We fitted a linear model with and without regularisation, doing a search for the optimal regularisation parameter λ. We looked at 300 λ-values in the range 10^{-8} to 10^5 and found that the best cross-validated R^2 score was achieved with $\lambda = 7390.7$, where the score was

$$R^2 = 0.803 \pm 0.005.$$

The results indicated that regularisation was beneficial, explaining $\sim 2\%$ more of the variance of the data compared to using no regularisation.

B. Feedforward Neural Network

We applied a FFNN with the rectified linear unit (ReLU) activation function in the hidden layers. The ReLU function has previously been shown to work well in deep learning architectures and to be a better model of how biological neurons compute information [18].

We required the hidden layers to all have the same number of nodes, and applied the Adam optimiser to find the optimal weights and biases. We searched for the ideal network architecture by varying the number of hidden layers and the number of nodes in each layer. We tested networks with up to three layers, with the number of nodes in each layer ranging from 10 to 1000. Additionally, we searched for the optimal regularisation parameter λ and initial learning rate η_0, looking at values ranging from 10^{-5} to 10^3 and from 10^{-4} to 10^{-1}, respectively.

We found that the optimal network architecture consisted of two layers of 100 nodes each. The optimal regularisation parameter was $\lambda = 10.0$, and the optimal learning rate was $\eta_0 = 0.001$. The best cross validated R^2 score was

$$R^2 = 0.886 \pm 0.002.$$

C. Support Vector Regression

We fitted a nonlinear SVR model with a radial basis kernel. We searched for the optimal regularisation parameter C and the optimal kernel parameter γ. The model for predicting the velocity in the x- and y-directions were fitted independently, but we applied the same hyperparameters to both. We found that the optimal regularisation parameter was $C = 3.54$ and $\gamma = 0.0002$, which yielded a cross validated R^2 score of

$$R^2 = 0.827 \pm 0.003.$$

In Figure 3 we have plotted the predicted velocities in the y-direction from the first 10 seconds of the experiment. The predictions from each of the algorithms are included in the plot, as well as the ground truth.

In Table I we have added the test scores from all the optimised algorithms. We report both the R^2 score and the RMSE. Note that the scores were computed on an unseen test set.

Table I. The R^2 and RMSE scores of the models with optimal hyperparameters, computed on an unseen test set.

Algorithm	Test R^2	Test RMSE [cm/s]
FFNN	0.892	1.96
SVR	0.830	2.46
Linear	0.802	2.65

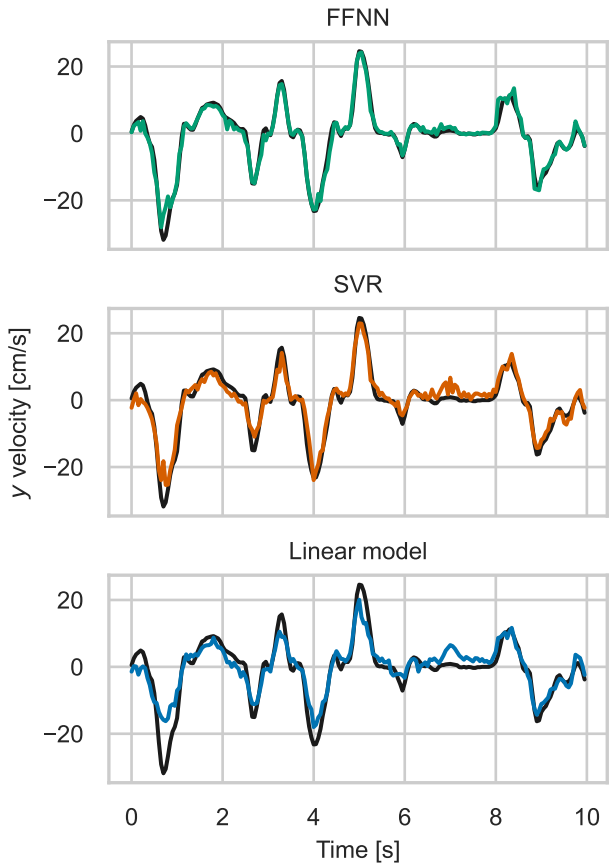

Figure 3. Predicted velocities (colored) and ground truth (in black) in y-direction. The values are based on the first 10 seconds of the experiment. The prediction from the different optimised algorithms is plotted from top to bottom (FFNN, SVR, linear model, respectively).

IV. DISCUSSION

A. Predictive Performance

We found that the FFNN outperformed both the linear model and SVR. Our results showed that the FFNN was able to explain 9.2% more of the variance in the data compared to the linear model. This is in line with the results that Glaser et al. found in [6], when comparing different machine learning algorithms on the same data set. They found that neural network models, such as the feedforward network and various recurrent networks, outperformed other machine learning algorithms. Similarly, we found that SVR outperforms the linear model, which is also in line with the findings of Glaser et al. However, in this case the nonlinear model was only able to account for 2.8% more of the variance than the linear model.

By optimising the regularisation parameter of the algorithms, which Glaser at al. did not do, we were able to improve the performance of both the FFNN and the linear model compared to Glaser et al. The R^2 score of the linear model was improved, going from 0.78 to 0.802. The best R^2 score of the FFNN model reported by Glaser et al. was 0.85. The best overall performance of the algorithms that Glaser et al. tested was achieved using a recurrent network algorithm, which had an R^2 score of 0.88. Using a FFNN with regularisation, we achieved an R^2 score of 0.892. This shows how important it is to account for regularisation, specially when fitting a complex model with many parameters. In this way, we can find the best trade-off between a low-variance and low bias model.

One of the challenges of this analysis, arose from the fact that it took a long time to train the nonlinear models. The FFNN used between 1 min and 15 min to train, depending on the learning rate and network architecture applied. The SVR algorithm took $\sim$ 2.5 min to train, and the linear model took $\sim$ 6 s. This somewhat limited the scope of the hyperparameter search. However, by using the `scikit-learn` library and parallelisation when doing the cross validation, we could still do an effective and thorough search of the hyperparameters.

In line with the findings of Glaser et al. we observed that the performance of the algorithms was fairly robust to changes in the hyperparameters. For example, when doing a grid search for the optimal FFNN hyperparameters we found that 70% of the hyperparameters we tested had an R^2 score > 0.75, mostly failing when using an initial learning rate of $\eta_0 = 0.1$. This, however, also shows that a thorough grid search should be strongly encouraged, as it can improve the performance of the algorithm, even if the algorithm generally performs well.

B. Considerations in Feature Engineering

When working with spike train data, it is common to construct feature vectors based on the activity of a population of recorded neurons, where some period of spiking history can be included. This is what we have done in this study. However, this approach demands that we extract the relevant features manually, and so we risk not using the data optimally for making predictions. For example, it is possible that a different choice of time resolution or time-delay could have further improved predictive performance. This problem could be alleviated by using recurrent network algorithms, where the algorithm automatically extracts the relevant features itself [5]. Specially in a field where it is desirable to record large populations of neurons at a fine-grained time scale,

this can be very useful.

Another problem of feature engineering that we have not considered in this study, is evaluating the relative importance of the features included. This is particularly relevant in the case where we want to know more about the relationship between the output and neural activity. Machine learning algorithms can be difficult to interpret, especially complex network models, and there are methods for revealing how much the various features contribute to a given prediction, both in terms of global predictive power and in terms of the individual predictions [19].

Finally, another interesting aspect to consider when working with neural data is how we assume neural activity encode information. In this study, we have assumed that the firing rate is the main encoder. Still, there are other hypotheses of what makes up the neural code. For example, it has suggested that the synchronous activity of pairs or ensembles of neurons encode information, or that there is important information encoded in the relative timing between spikes [20]. Such considerations could inspire multi-modal modelling, where neural activity from different biological scales (i.e. from individual electrodes in neurons to full brain imaging) are combined to capture multiple aspects of the neural code [21].

All the mentioned problems related to feature extraction and evaluating feature importance motivate further investigation in the future.

V. CONCLUSION

In this study we applied nonlinear and linear machine learning algorithms for predicting movement velocity from neural activity recorded in the primary motor cortex. Specifically, we applied feedforward neural networks, support vector regression, and a linear model. We found that the neural network had superior predictive performance, and that we were able to outperform the results from a similar study done on the same data set by further optimising the hyperparameters.

We observed that regularisation is particularly impactful, improving the performance of both the linear and nonlinear algorithms. We also observed that the algorithms' performance was fairly robust over a range of hyperparameters. Still, the fact that we were able to improve performance for all algorithms, explaining about 2% more of the variance relative to a less optimised algorithm, shows that a thorough grid search is beneficial and worthwhile.

Finally, we discuss some of the limitations of the methodology, by highlighting that there are many possibilities when selecting the relevant features in neural decoding. These limitations involve what hypothesis of neural coding is applied in feature engineering, as well as the manual selection of features with respect to time resolution and time-delay. We also note that, in computational neuroscience, it is desirable to be able to understand the relative importance of the features to gain a better understanding of the neural code. Our study focuses exclusively on improving predictions, and not on interpretability.

[1] N. Ahmadi, T. G. Constandinou, and C.-S. Bouganis, Decoding hand kinematics from local field potentials using long short-term memory (lstm) network, in *2019 9th International IEEE/EMBS Conference on Neural Engineering (NER)* (2019) pp. 415–419.

[2] R. Quian Quiroga and S. Panzeri, Extracting information from neuronal populations: information theory and decoding approaches, Nature Reviews Neuroscience **10**, 10.1038/nrn2578 (2009).

[3] J. I. Glaser, A. S. Benjamin, R. Farhoodi, and K. P. Kording, The roles of supervised machine learning in systems neuroscience, Progress in Neurobiology **175**, 126 (2019).

[4] J. A. Livezey and J. I. Glaser, Deep learning approaches for neural decoding across architectures and recording modalities, Briefings in Bioinformatics **22**, 1577 (2020).

[5] P. Szabó and P. Barthó, Decoding neurobiological spike trains using recurrent neural networks: a case study with electrophysiological auditory cortex recordings, Neural Computing and Applications , 2825 (2022).

[6] J. I. Glaser, A. S. Benjamin, R. H. Chowdhury, M. G. Perich, L. E. Miller, and K. P. Kording, Machine learning for neural decoding, eNeuro **7**, 10.1523/ENEURO.0506-19.2020 (2020).

[7] P. Dayan and L. F. Abbott, *Theoretical Neuroscience: Computational and Mathematical Modeling of Neural Systems*, 6th ed., edited by L. F. Abbott (MIT Press, 2001).

[8] I. Park, S. Seth, A. Paiva, L. Li, and J. Principe, Kernel methods on spike train space for neuroscience: A tutorial, IEEE Signal Processing Magazine **30** (2013).

[9] J. I. Glaser, M. G. Perich, P. Ramkumar, L. E. Miller, and K. P. Kording, Population coding of conditional probability distributions in dorsal premotor cortex, Nature communications 10.1101/137026 (2018).

[10] D. H. Hubel and T. N. Wiesel, Receptive fields of single neurones in the cat's striate cortex, The Journal of Physiology **148**, 574 (1959).

[11] F. Liu, S. Meamardoost, R. Gunawan, T. Komiyama, C. Mewes, Y. Zhang, E. Hwang, and L. Wang, Deep learning for neural decoding in motor cortex, Journal of Neural Engineering **19**, 056021 (2022).

[12] V. Gilja, P. Nuyujukian, C. Chestek, J. Cunningham, B. Yu, J. Fan, M. Churchland, M. Kaufman, J. Kao, S. Ryu, and K. Shenoy, A high-performance neural prosthesis enabled by control algorithm design, Nature neuroscience **15**, 10.1038/nn.3265 (2012).
[13] W. Wu, M. Black, Y. Gao, M. Serruya, A. Shaikhouni, J. Donoghue, and E. Bienenstock, Neural decoding of cursor motion using a kalman filter, in *Advances in Neural Information Processing Systems*, Vol. 15, edited by S. Becker, S. Thrun, and K. Obermayer (MIT Press, 2002).
[14] A. Jakobsen, D. Haas, N. Taugbøl, and V. Ung, Exploring the transition from regression models to neural networks (2022).
[15] A. J. Smola and B. Schölkopf, A tutorial on support vector regression, Statistics and Computing **14** (2004).
[16] C. M. Bishop, *Pattern Recognition and Machine Learning (Information Science and Statistics)* (Springer-Verlag, Berlin, Heidelberg, 2006).
[17] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay, Scikit-learn: Machine learning in Python, Journal of Machine Learning Research **12**, 2825 (2011).
[18] X. Glorot, A. Bordes, and Y. Bengio, Deep sparse rectifier neural networks, in *Proceedings of the Fourteenth International Conference on Artificial Intelligence and Statistics*, Proceedings of Machine Learning Research, Vol. 15, edited by G. Gordon, D. Dunson, and M. Dudík (PMLR, Fort Lauderdale, FL, USA, 2011) pp. 315–323.
[19] C. Molnar, *Interpretable Machine Learning* (2018) https://christophm.github.io/interpretable-ml-book/.
[20] S. Thorpe, A. Delorme, and R. Van Rullen, Spike-based strategies for rapid processing, Neural Networks **14**, 715 (2001).
[21] H. Y. Lu, E. S. Lorenc, K. Zhu, H., S. J., C. J., Xie, P. N. Tobler, Watrous, O. A. J., A. L., J. Lewis-Peacock, and S. R. Santacruz, Multi-scale neural decoding and analysis, Journal of Neural Engineering **18**, 10.1088/1741-2552/ac160f (2021).