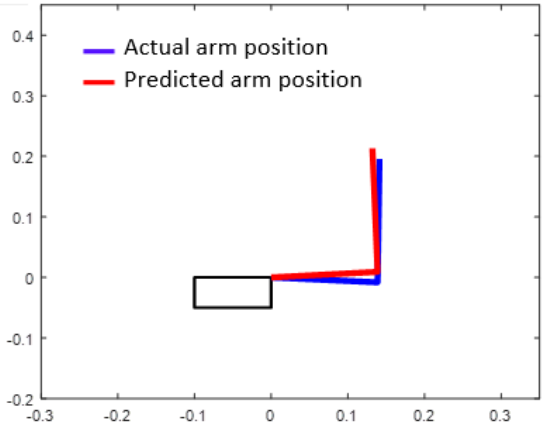
Brain-Computer Interaction – HOMEWORK 3

Introduction

During this exercise session, you will explore a variety of issues related to model and experiment design. This exploration will be done in the context of a real monkey data set that has been collected and analyzed by Dr. Nicho Hatsopoulos of the University of Chicago (Fagg et al. 2009).

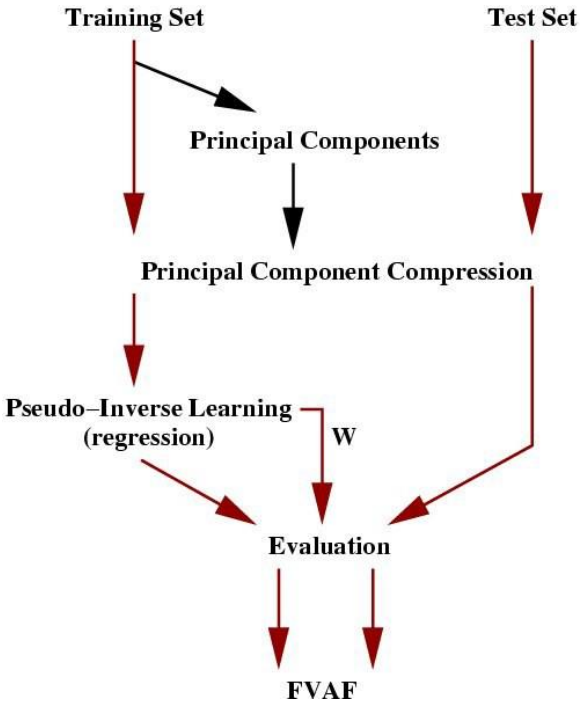
This data set contains approximately 13000 separate samples in which neuron activation patterns are paired with arm motion variables.

The software that is provided for this laboratory exercise will allow you to configure data sets for training and testing of a model and to evaluate the performance of the model that results from this process. The models can be configured to predict any number of observed arm motion variables, including Cartesian position, velocity, and acceleration (of the wrist), and joint position, velocity, acceleration, and torque (see figure below for joint prediction).



The **figure below** outlines the data flow through our lab software. The red arrows correspond to components that will be used throughout the set of tasks; the black arrows (dealing with principal components) will be used for tasks 4 and above. In general, the first step in using the model is to select the training and test sets. For tasks 1-3, the training set is then used to derive a model (described by W see Matlab Workspace parms.net.W). This "learning" process makes use of the Moore-Penrose pseudoinverse method as a way of performing regression. Once this model is constructed, we can then evaluate the performance of the model on both the training and test sets.
In order to facilitate comparison between the different models, performance is always reported in terms of the **fraction of variance accounted for (FVAF).** This measure ranges from 1 (perfect prediction) to negative infinity. Typically, we will see positive values for this metric (although there will be a few cases where we will observe negative values).
For tasks 4-7, we will make use of Principal Component Analysis (PCA) as a technique for preprocessing the neural activity vectors. This step has the effect of compressing

these activity vectors to a smaller-dimensional representation. Regression and subsequent evaluation will be performed in these modified representations.



Task 1: Basics Initialization

Start matlab and set the current working directory to **the provided folder.** Initialize the BMI module:

>> init

This will load the full data set and initialize the internal data structures. The loading process will take a bit of time.

Obtaining Status

At any time, you will be able to obtain the status of the experimental model:

>> status

This command will tell you the set of data folds that have been loaded, which are currently being used as training and test folds, the number of principal components that are being used for compression, the type of prediction, and the performance on the last training run. Note: when **PCA compression** is set to **0,** then no compression is performed.

Selecting Training/Test Sets

The set of folds used as training and test sets can be set with the following commands:

>> set_train([fold list])

>> set_test([fold list]) - Where **fold list** is a scalar or a vector of indices of available folds.

Examples:

>> set_test(5)
>> set_train([1,2,5])
>> set_test(1:5)

Training

Training of the predictor is initiated using the following command:
>> train

The time required to perform this operation will depend upon the size of the training set, the degree of PCA compression, and the clock rate of your computer.

Following training, this command will report the predictor's performance for both the training and test sets. Those value are also stored into **parms.net.fvaf_train** and **parms.net.fvaf_test**.

For each set, two values are reported: one for each dimension of prediction. In the case of Cartesian predictions, these numbers correspond to the X and Y dimensions; in the case of joint-related predictions, they correspond to the **shoulder** and **elbow angles**, respectively. Prediction possibilities include Cartesian position: **'X'**, velocity: '**dX'**, acceleration: **'ddX'**, joint angle: **'theta'**, joint angular velocity: **'dtheta'**, and joint angular acceleration: **'ddtheta'**.

The numbers represent the **fraction of variance accounted for** each prediction. A FVAF of 1 corresponds to a perfect prediction; 0, a poor prediction; and values less then 0 are very poor predictions (and are usually indicative of a problem with overfitting of the training set).

Visualization

Following training, you can visually compare the prediction results to the true dataset:
>> newwindow([figure #], [display type])
Where **figure #** is a positive integer, and **display type** is a string indicating the information to be displayed (this string can be either 'train', 'test', 'input_train', 'input_test'). When 'train' or 'test' are provided as parameters, the window will display the actual and predicted quantities for the training and test sets, respectively. 'input_train' and 'input_test' will display the neuron activation patterns used as inputs into our linear function.
The original data is shown in blue; predictions are shown in red. Each display window contains a set of navigation buttons that allow you to increase/decrease time, to zoom in/out, and to redraw the display.

You may open as many figure windows as you wish (simply supply a different **figure #** for each).
Example:
>> newwindow(1, 'test')

Perform the following:

• Pick one fold as training set
• Pick a different fold as test set
• train

Open widows for both train and test.
Q: why does the performance differ so significantly for the training and test set?
Comment on the difference between the predictions for these two data sets.

Task 2: Training Set Effects

Any time we construct predictors, we are faced with the question of how much training data is enough so that we end up with a predictor that can perform well in novel situations. In this task, we will ask this question. Perform the following:

Perform the following:
• Pick one fold as training set
• Pick a different fold as test set
• Train. Record the resulting performance (both the test and training performance).

• Increase the number of folds used as training set to two
• Test on the same fold as before
• Train. Record the resulting performance

Q: How does the training set performance change as we move from one to two training folds? Why?
Q: How does the test set performance change as we move from one to two training folds? Why?

• Continue by increasing by one the number of training set until the training set reach the maximum number of folds available and record these results.
• Report in one single figure the curves for all the predictions (i.e. train shoulder, test shoulder, train elbow, test elbow). Use plot function in matlab (see Figure 1 as template).

Q: Based on these results, which is the number of folds you would use for the train set? Which considerations you make in performing such choice?

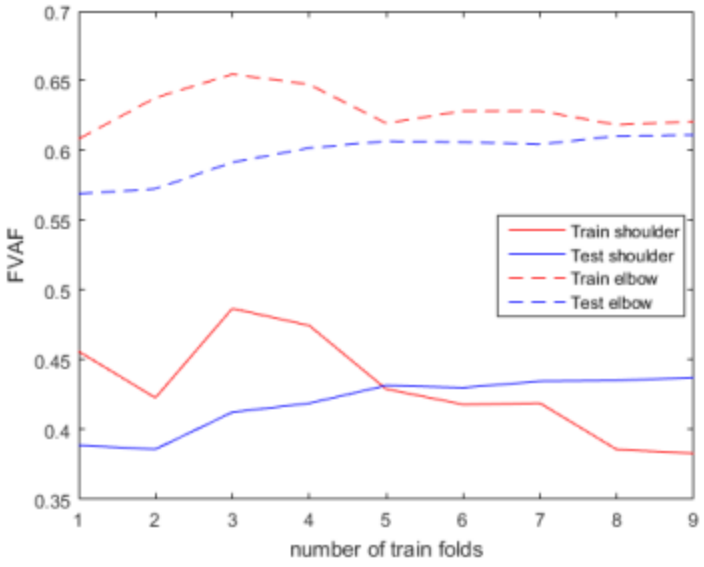


Figure 1

Task 3: Comparing Output Predictions

One of the ongoing debates in the neurophysiology of control is: **what** is the meaning of the firing rate of a single neuron? There are many different ways to describe the motion of the arm (e.g., Cartesian/joint location/velocity/acceleration, or joint torque). A common way to approach this debate is to assume that the predictions must be made in one of these coordinate frames, to construct a predictor of motion in each of the coordinate frames, and then to compare the performance of the resulting predictors. You have already evaluated a set of torque predictors

• Set the predictor type to **Cartesian** with the following command:

```
>> set_type('X')
```

• Pick two folds as training set and one fold as test set

• Train. Record the resulting performance

Q: Examine the predictions graphically (both training and test sets). How does it compare with the corresponding **torque** experiment?

• Select another predictor of your choice (possibilities include Cartesian velocity: 'dX', acceleration: 'ddX', 'theta', 'dtheta', and 'ddtheta')

• Train. Record the resulting performance

• Plot on one single figure all results according to the predictor type (Cartesian, torque, velocity, acceleration), for every type of predictor (train shoulder, test shoulder, train elbow, test elbow). Use bar function in Matlab (see Figure 2 as template).

Q: how does this performance compare with the other two experiments?
Q: what could you conclude from these observations?

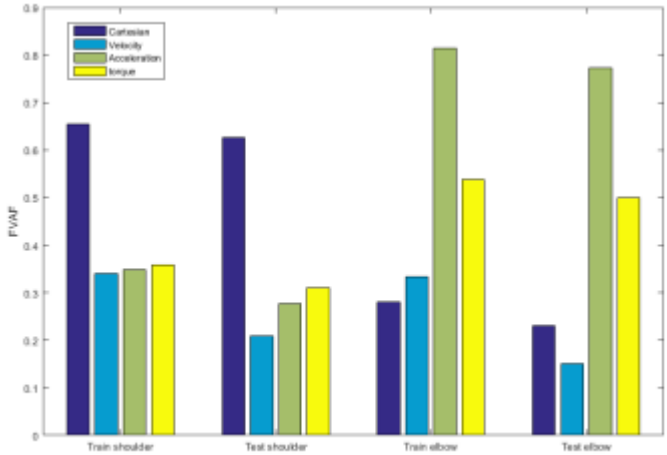


Figure 2

Task 3.1: Predicting Joint Motion
When the prediction type is set to **theta,** you will be able to visualize the actual and predicted arm motions in real time. Use the following commands to do so:
>> animate('train')
>> animate('test')
As in the static displays, blue represents the actual, observed configuration of the arm; red represents the corresponding predicted configuration.

Task 4: Principal Component Preprocessing

As stated to date, our prediction problem is one of predicting two variables from 960 variables (the latter of which describe the recent history of activation levels for a set of cells). Each training fold consists of approximately 1100-1500 individual predictions (which correspond to different points in time). As we are trying to solve for our parameter matrix (W), one way to state this problem (assuming one fold as a training set) is that we have a total of **~1100 equations and 960 unknowns.** While this can be an over-constrained problem, in practice, many of our equations are redundant with one-another (in other words, they are linearly dependent). What results is essentially a very unconstrained problem that can lead to wild overfitting of the training set (by the pseudo-inverse method).

By **overfitting** we mean that although performance can be high for the training set, generalization to an independent test set can be very poor (this is seen as FVAF measures that are less than 0).

The redundancies in the data set are due in part to a couple of factors. First, two different neurons can be highly-correlated with one-another. This means that we could actually encode their current firing rates using a single variable (as opposed to two). Second, we are representing the firing rate of a single neuron using 20 variables (each one corresponding to different time delays). These variables are clearly correlated (just shifted in time relative to one-another). The effect of either form of correlation is that our 960-dimensional input vectors do not span the entire 960-D space, and instead exist in a lower dimensional manifold of this space.

One way to address the overfitting problem is to remove the redundancies in the input side of the data set before we perform the regression step. Here, we will use **Principal Component Analysis** to remove these redundancies. PCA is an algorithm that identifies the primary dimensions of covariation of a set of variables. With PCA, we can project our individual, 960- dimensional input vectors into some N-dimensional space (where N < 960), while still preserving much of the information encoded in the original vectors. In doing so, we hope to remove any small variations that are orthogonal to the manifold (these variations primarily encode noise), and thus eliminate the need for the regression method to attempt to account for these small variations.

In our implementation, you can select the degree of PCA compression using the following command:

```
>> set_pca([N dimensions])
```

Where **N dimensions** is the number of dimensions that you wish to compress to. For our particular data set, N_dimensions = 960 represents no compression, and N_dimensions = 2 represents a significant degree of compression (each vector describing motor cortex activity is compressed from 960 to 2 variables). Note that for higher degrees of compression (lower numbers), the regression process will be faster (we are taking the pseudo-inverse of a smaller matrix).
Also note that:
>> set_pca(0) is equivalent to **no compression.** Perform the following experiment:

Perform the following:
• Pick two folds as training set and one fold as training set
• Set the prediction problem to **torque**
• Select a high PCA compression level (e.g., >> set_pca(1)).

Q: how does this compare to the uncompressed case?

• Increase the number of principal components
• Train. Record the resulting performance.
• Repeat and plot the FVAF as a function of the number of components (Hint: if the computation takes too long, use steps of 1 from 1 to 50 components and then increase by steps of 50). Report the curves for all the predictions, i.e. train shoulder, test shoulder, train elbow, test elbow, on one single figure). Use Figure 1 as template.

Q: what happens to the performance when you apply PCA?
Q: As the number of principal components increases, at what point do you have a maxima in the test set performance?
Q: why is there a maxima in the test set performance?
Q: How does this compare to the case where there was no compression?

Task 5: Proprioceptive Inputs

Up to this point in time, we have assumed that the torques we are predicting are purely functions of neuron activity. However, we know from the anatomy that between the motor cortex and the muscles, there are low-latency feedback loops that bring state information back into the computation. We model this here by appending the joint state vectors (joint orientation and joint velocity) to the 960-dimensional neuron activity vector (to give us 964 dimensions).

In the simulation, this appending is accomplished by:
>> set_proprioceptive(1)
In order to turn this option off, you can use:
>> set_proprioceptive(0)

Perform the following:
• Select a prediction mode of **torque**
• Select **no** compression PCA
• Pick one fold as test set
• Plot the FVAF as a function of the number of folds in your train-set (from 1 to 9 folds) for both models (with and without proprioception). Report one figure for the shoulder (curves: train_proprioceptive_off, train_proprioceptive_on, test_proprioceptive_off, test_proprioceptive_on) and one figure for the elbow (same curves).
Use Figure 1 as template.
Q: How does this change to the model affect the performance in both the training and test sets?: Why?

Note that this "proprioceptive mode" only makes sense for predictions of torque or accelerations.

Task 6: Varying Neural Delays

In the models that we have constructed so far, the state of each neuron is described with a total of 20 different features. Each feature represents the firing rate of the neuron during a 50 millisecond time period. So, with the 20 features, we represent the neuron's state from the current time, back to 1 second prior to this time. A critical question is how much of this information is necessary to make an arm motion prediction now.
You are able to select the amount of neural history that is used in the prediction process using the following command:

>> set_delays(vec)
where **vec** is a vector of time indices that can contain the values 1 through 20. For example, you can specify that only the most recent 50ms be used with the following command:
>> set_delays(1)
And the most recent 100ms (two features per neuron) is selected with:
>> set_delays(1:2)
And the most recent 500ms (ten features per neuron) is selected with:
>> set_delays(1:10)
Finally, you can return the configuration to the original 1 second with:
>> set_delays(1:20)

Perform the following:
• Select a prediction mode of **torque**
• Set **no** PCA compression
• Pick two folds as training set
• Pick one fold as test set
• Vary the delay from 1 to 20 (with several others in between) and train.
• Plot FVAF as a function of the delay.
Report all curves on one single figure: train shoulder, test shoulder, train elbow, test elbow. Use Figure 1 as template**.**

Q: As the time period is increased, how does the performance change in both the training and test sets?
Q: At what point does the performance reach a maxima?
• Q: As the time period is increased, how does the performance change in both the training and test sets?
Q: At what point does the performance reach a maxima

[BONUS] Task 7: Optimizing the model

The prediction models we constructed in this exercise have several parameters and hyperparameters. We define parameters as elements of the model that are learnt from the data itself. In a regression model, for example, the coefficients of the regressor are parameters because they are set by optimizing some measure of error (e.g. the mean squared error) between the actual and the predicted values. Conversely, we define hyper-parameters (HPs) as elements of the model that cannot be learnt from the data itself.

In our exercise, examples of HPs are: the number of folds to be used for training the model, the number of features (e.g. the number of PCs), the type of the predicted variable (torque, speed etc), whether to include proprioceptive inputs or not, etc…

Across the tasks of this exercise, we have been varying one HP at a time (while fixing all the others), and checked how this variation affect the results. In reality, if one wanted to optimize the model with respect to all the HPs, this procedure would not be optimal.

Q: Can you guess why?
Q: Do you know any technique that would allow to correctly optimize all the HPs?

References

Fagg, A. H., G. W. Ojakangas, L. E. Miller, and N. G. Hatsopoulos. 2009. “Kinetic Trajectory Decoding Using Motor Cortical Ensembles.” *IEEE Transactions on Neural Systems and Rehabilitation Engineering* 17 (5): 487–96. https://doi.org/10.1109/TNSRE.2009.2029313.