Bioe 1586
Homework: Neural Prosthetic Decoding

In this assignment you will implement two different neural prosthetic decoders: a linear filter, and a Bayesian classifier. The first is a continuous decoder that allows you to estimate kinematic parameters of the hand movement from the activity of a population of neurons. The second is an endpoint decoder that predicts the monkey's reach goal from the population of neural activity.

Part 1: Continuous decoding

You will estimate hand position using a simple continuous decoder: the linear filter. Load the dataset *continuous.mat* (data courtesy of Nicholas Hatsopoulos at the University of Chicago.) The dataset contains two variables: *kin* is the position of the hand (x and y coordinates) recorded over one hour, and sampled every 70 ms. *rate* is the averaged firing rate (over the 70 ms bins) of the activity of 42 neurons recorded in primary motor cortex.

1. First, visualize the behavioral data. The monkey is performing a "random target tracking" task where he reaches toward one target after another as they appear throughout the recording session. The function *comet* gives a nice way to replay the behavioral performance. Generate a figure showing the behavioral data. Choose a plotting format for the data that conveys the essence for the behavior without being too crowded.

2. Now compute the linear filter coefficients. The linear filter is Y = XA, where Y is 2D kinematics, X is neural activity (42D), and A is the coefficients of the linear filter, which must be solved for. Note that this is just a matrix version of y=mx+b, and the linear filter is essentially just multidimensional linear regression. Append a vector of 1's to the X matrix to provide those B offsets. Solve for A to build your BCI decoder, as we discussed in class on March 23rd.

Note that the linear filter used in most experiments includes many temporal lags between behavioral and neural data. That is, neural activity can drive cursor movement with several different time lags. However, such models include a lot of parameters, but they are not conceptually different from the model with one lag. For now, simplify things by using just one lag. This can be accomplished simply by shifting the behavioral data. Time-shift the kinematic data by two time bins (140 ms), which is a pretty good estimate for the latency from M1 activity to hand movement.

3. Observe and quantify the results. Test the decoder's performance by multiplying the neural data by the coefficient matrix A that you just found. Plot the results in an informative format. Play with the data range if you want to zoom in or out of the results. Does the decoder do better at reconstructing the small details of the movement or the "big picture"?

Quantify performance somehow. Consider measuring the mean squared error between the actual reach and the reconstructed reach. This is a nice "sanity check" on your code, but since the training and testing data are the same, obviously you could not use this for a real-time application. In actual experiments, performance is quantified with things like success rate, time-to-target, or straightness of the path.

4. In a real-time neural prosthetic application, the training data are separate from the test data. Load continuous2, another set of neural and kinematic data, and attempt to reconstruct the new kinematic data from the new neural data, using the coefficient matrix A you learned from the training data. Plot the reconstructions.

Again quantify the fit with correlation and mean squared error analyses. Compare these values to the ones you got when you trained and tested on the same data. Is performance better or worse? Does that make sense to you?

5. Try various lags (a straightforward adjustment to the code.) What if you use time lags of 0, 70 ms, or 210 ms? What time lag yields the best reconstruction? What if you use an anti-causal time lag (that is, behavior leads neural activity, rather than lagging it.)

If you want to build a full model that allows multiple time lags, it's a simple adjustment to the data matrix - include columns for each time lag. Does that yield better performance? How many time lags give the best results? Do you think including more time lags always yield better reconstructions?

Part 2: Discrete decoding

You will use a Bayesian classifier to predict the endpoint of a reach, given the spike counts of a population of neurons. Doing so involves a training and a test step: For training, you measure the likelihood of neural activity given a particular target. For testing, you take in measurements of neural activity, and from them predict the target.

Neurons are noisy; when you present the same target repeatedly, a neuron will discharge a slightly different number of action potentials. This variability in the response of a neuron is generally assumed to be well-characterized by a Poisson distribution. That means that we only need to estimate a single parameter, lambda, for each combination of neuron and target. Lambda can be estimated simply as the mean spike count observed over the repetitions of that reach. (Proving this takes a little bit of math, but the statement itself should be pretty intuitive.)

Since with a discrete decoder, you do not estimate the entire trajectory, just the endpoint, the quantification of the decoder's performance is simple: you can generate a "confusion matrix" that plots the actual target location against the decoded target location. This allows you to see how often the decoder is correct, and when errors are made, you can see if it's off by much.

The following steps take you through the process:

1. Load *spikeCounts.mat* to create a workspace variable SpikeCounts (data courtesy of Krishna Shenoy, Stanford University). It is a three-dimensional matrix where the dimensions are [repetitions, neurons, targets]. Get a feel for the tuning of the cells by plotting the average spike count for the 5 targets for a handful of cells.

2. Since the training and test data need to be distinct, we will use a technique called *leave-one-out crossvalidation*. In it, a single trial is set aside as the test data. The decoder is trained using all of the other data, then its performance is tested on the trial that was set aside. Every trial is selected in turn to be the test data. In this way, we can get the most out of our data and perform (numReps * numTargs) tests of the decoder.

3. For each test of the decoder, follow these steps:

A. Train the decoder using all the trials but the one you set aside.

Training is a simple matter of computing the mean spike counts for each neuron and target combination. This is the asset of the Poisson assumption.

B. Test the decoder on the trial you set aside.

C. Store the actual target location and the decoded target location so you can quantify the decoder's performance. You can put these into a matrix of size (numreps * 2) where the first column is the actual target and the second column is the decoded target.

4. Plot the confusion matrix. You can use my "plotconfusion.m" or write your own. Build a 2D histogram with actual target on the x axis and decoded target on the y axis. The values of that histogram are the number of occurrences of each combination of actual versus decoded target. Plot that histogram as a colormap. Turn in this figure. What is the decoder's overall percent correct? What are your thoughts on how you could improve the endpoint decoder's performance?