Welcome to nexusbmi, a repo for doing BMI with the Activa PC + S, Nexus-D streaming capabilities, and a custom Mario Game (see Khanna 2015).

To train decoder:

* Record any task that has ‘dat’ structure saved
* Select source to use as features
* Default decoder computation (utils > make\_decoder.m)
  + Feature extraction:
    - Power method – use raw power channel data (skip empty indices)
      * Lower\_lim, upper\_lim automatically set to zero
    - Time domain method – extract\_freq\_feats\_from\_td.m
      * Pulls up interface to test out different features
      * Test frequency bands with most variance (presumably you are training from a movement task and are looking for beta variance).
        + Returns sum(Pxx(low\_lim:high\_lim, :),1), sums over frequency band at each time point. Note – no normalization
  + Decoder training:
    - Simple method
      * Quarter\_step is defined as difference between 25th-50th percentile on features extracted
      * Mean is 50th percentile
    - KF method (init\_KF.m)
      * Requires features, decoder, targ\_pos
        + Targ\_pos can be used from task (if seeding from actual NF task), or inferred (via percentiles)
      * Normalize features:
        + Square-root (to normalize power computation), then subtract mean
      * Compute KF:
        + Y = neural;, X = [target\_pos + noise ~N(0, .01), constant];

Size: 1 x bins

* + - * + A, W computed empirically
    - Save\_trained\_decoder.m
  + Current decoder structure:
    - Feature\_band: [lower\_lim, upper\_lim]
    - Mean
    - Quarter\_step
    - Source: ‘nexus…’
    - Method: ‘simple’

Code Flow:

This codebase is modeled off of the Carmena Lab brain-python-interface (bmi3d) repository which can be found at [https:/github.com/carmenalab/brain-python-interface](https://github.com/carmenalab/brain-python-interface). Bmi3d is intended for streaming neural data from electrophysiology recording setups to execute brain-machine interface tasks at low latencies. Much of the code structure and task flow principles from bmi3d is borrowed here.

**The interface**:

nexusbmi starts when the user enters ‘mini\_bmi’ at the command line. This summons a GUI that is used to start tasks. In order for a task to proceed, a task must be selected from the drop-down menu in “Task Params”, a decoder must be selected from the drop-down menu in “Task Control”, and a Neural Source must be checked. Ensure that the Extractor Params (esp. Data Chan Idx) and Extractor type are correct.

**Task Control from the Interface**

* **GO:** starting the task occurs by pressing the ‘Go’ button
  + A call is made to init\_task.m where:
    - java files are added to the user path (can only happen once per session, hence the global variable),
    - Reward sounds and images are loaded into the “handles” structure
    - Task display is initialized
    - Data files are initialized
    - Decoder is loaded
    - Neural source is cued up
    - Arduino is started (if the Arduino check has been made)
  + Then the task proceeds through the **run\_task.m** flow
* **STOP**: made by pressing the stop button
  + During the task, after every iteration a call is made to **update\_handles\_from\_gui.m** to check if anything in the GUI has changed (ex: assist, stop button pushed, filter, timeout time).
  + If the stop button has been pushed, **cleanup\_task** is executed

**The Full Task Loop: (run\_task.m)**

* First we check if there’s neural data available (if the last time we checked is greater than loop time). For all nexus applications, loop\_time is 0.4 seconds, and probably should be defined in the extractor, not the task (TODO)
  + If it’s time to pull neural data we pull neural data (handles.neural\_source is another object and obs.get\_neural is the object method to get neural data)
  + Then the feature extractor object is used to extract features (handles.feature\_extractor > extract\_features)
  + Then absolute time at which the neural data was pulled (our best estimate of the time it occurred) is recorded (T).
  + We can calculate cursor position (handles.decoder.calc\_cursor)
  + The task state, display, data file are then all updated (as they would be even if neural data wasn’t available).
* Notice how the neural interface, feature extractor, decoder are all objects. This means the run\_task file can be used regardless of which specific sources, feature extractors, and decoders are used.

**Decoders:**

**Simple:**

* Takes mean of features (not suited for many features)
* Obj.run\_decoder just subtracts the mean and divides by the std.
* Then if there’s a LP filter or assist those are computed
* Note: cursor position gets saved as handles.window.cursor\_pos(2) whereas the decoded position (from the output of the decoder) is saved as obj.decoded\_position (outside of this fcn it is handles.task.decoded\_position).

**KF:**

* Normalizes features (sqrt, then subtract mean from decoder)
* Same as above but time and measurement updates are run
* CLDA – RML (Dangi 2014) is also been implemented but unsuccessfully used (probably because there’s no tuning in beta band).

**Feature Extractors:**

**Nexus Power Extractor:**

* Only one used thus far (accel extractor was used to test decoder implementations)
* Initialization requires ‘extractor params’, which appends the ‘f\_range’ from the decoder to [0, 200], [0, 100]. This is obj.f\_ranges in the feature extractor. Obj.task\_f\_ranges is just ‘f\_range’ from the decoder. ‘extractor\_params’ also carries the ‘used\_chan’ from the chan\_data\_ix from the task interface.
* So, remember:
  + Obj.f\_ranges contains all features being computed (t.d. only)
    - Obj.n\_features is the number of obj.f\_ranges
  + Obj.task\_f\_ranges contains feature band being used for the task (t.d. only)
  + Obj.range\_inds has the indices needed to compute each bands’ power
  + Obj.chnfeat\_index is a vector of the ‘used\_chan’ number , repeated obj.n\_features times (e.g. [2, 2, 2]’, WHY?)
  + Obj.ftfeat\_index is something (WHY?)
  + Obj.task\_indices\_f\_ranges is a binary vector indicating which of obj.f\_ranges is the obj.task\_f\_ranges
* Get features utilizes the above to provide features.(obj.domain), a structure with the computed features

**Tasks: Computing task-state**

All tasks are classes (classes > tasks ), and each task is instantiated as an object during the running of a task. Each task has finite state machine that defines when it transitions from one task-state to another. These are in ‘FSM’. Importantly, these variables MUST be defined correctly:

FSM: rows constitute task transitions, column1 is state you are transitioning from, column2 is state you are transitioning to, and column 3 is the function used to assess if the transition occurred

state\_name\_array: cell array with list of states possible

state\_ref: 2D cell array where state\_ref{1} corresponds to all the transitions where state\_name\_array{1} is in the first column. State\_ref tells the tasks which transitions to check given that the task is in a particular state

Each task has a cycle function, where all the relevant state transitions are checked, and any other task functions occur that happen every iteration.

**Specific Task Definitions:**

* Movement cues:
  + Wait, beep, Wait, beep, etc.
  + Define go\_cue mean and go\_cue std. through holdMean and holdVar on the task interface
* Target task: (TO UPDATE, need to rm Arduion stuff)
  + Wait, middle target, hold periph target, hold reward
  + Target generator is middle, periph, middle, periph etc.
  + Params:
    - Hold time + hold std controlled from the interface
    - Timeout time controlled from the interface
    - ITI is hard coded = 0.1;
  + Right now gets a reward for getting back to the middle
* Finger tapping: (TO UPDATE)
  + Same as target task, but with tapping after each target except center target
* Target\_touch\_task (deprecated)

**Hardware:**

**Arduino**

**Connected:**

* Arduino Uno with shield is loaded with the .ino file: utils > faster\_serial\_comm
* Current configuration uses parallel pool to acquire Arduino data into a txt file that is continuously being written to ( run\_slim\_arduino.m)
* Slim\_arduino uses a faster baudrate (115200) and reads from the serial port using fscanf
* Synchronization occurs via the computer clock
* Current sampling rate is ~20 Hz
* Bitwise comm may be needed if adding more sensor values
* The Arduino knows when to stop comm transfer because the task writes to a file (decoder > shared\_process.txt)

**BT (deprecated):** BT was too slow for MATLAB to acquire sensor data at a reasonable rate (can see classes > sensor\_interface > bt\_ard).

**Database:**

**Datafiles:** Currently there are 3 data files that get saved from every task:

1. H5 file 2) dat file 3) txt file

Why? The h5 file is saved as the task runs to prevent the loss of data if the task crashes and the cleanup function hasn’t been reached yet. The dat files is needed for decoder training (as of now). It’s pretty redundant but it’s nice to have both types of files. Naming is consistent for all of them, but it is possible to OVERWRITE old data (ex. If data naming as based off of the .dat file and there’s only dat files a -c but there’s an h5 files that’s ‘d’, then the name will converge on ‘d’ and the h5 d file will be either overwritten or confusingly appended to. (TO TEST).

Graphics

Media

Analysis

Config

Nexus Utils