Neural Activity Regression Framework (NARF)

Ivar Thorson

December 5, 2012

This document was written to clarify and document design decisions made regarding the Neural Activity Regression Framework (NARF) implemented in the Brain, Hearing and Behavior Laboratory of the Oregon Health and Science University (OHSU), under the supervision of Prof. Stephen David. As such, it is intended for developers rather and is very much a work in progress.

1 Problem and Approach

1.1 The Problem

There are many possible mathematical models which could be used describe the spiking activity of a neuron given some sensory stimulus. In our case, we will be considering mostly models of neurons in the auditory cortex of a mammal, usually a ferret, and usually near the 'A1' region. Neurons in this region of the brain often respond preferentially to a particular auditory frequency. For example, some neurons may spike often whenever a sound with accoustic energy near 1.2kHz is heard. Another neuron may spike frequently whenever a 28kHz tone is heard. Our goal is to play recordings of various sounds, measure neural activity, and attempt to discover as much as we can about the neuron(s) we are studying.

Mathematically, the problem is that of developing a mathematical model which can accurately predict how a neuron will fire when a new, as-yet-unheard sound is played. This is a classic inference problem, in which we seek to infer which system produced the neural response that we measured. If the model's prediction closely matches the measured response of the neuron, then we have learned something significant about the neuron. The better our ability to predict the neuron's activity, the better our understanding of it, and the more we may learn through experiment about how neural activity is affected by various factors. Our ability to learn about the dynamics of neurons through experiment is thus closely related to our ability to model each neuron.

Fortunately or unfortunately, while the neuroscientist community has developed a plethora of mathematical models which can describe this behavior, there remain few guidelines on which model should be used or how model parameters should be found. The difficulty for most scientists is in the implementation

of existing models; it is often a significant amount of work to adapt another person's model to your needs, and significantly more work to develop one yourself. Ideally, we would have some way of fitting several different models to the measured activity of the neuron, so that we might quickly identify the characteristics of the neuron under study. It would also be advantageous to be able to recombine parts of the model which were particularly useful in describing neural activity such that variations on successfully predictive models could be quickly examined.

1.2 NARF's Approach: Pure Function Composition

While there can likely never be a completely general solution to the neural model fitting problem, borrowing well-developed concepts from mathematics can at least get us part of the way there. If we consider each aspect of a mathematical model to be a **pure function** f_1 which accepts an argument x_1 and returns a value x_2, we can write it as

$$f_1(x_1) \rightarrow x_2 \tag{1}$$

The notion of a 'pure' function is simply that it will always produce the same result when given the same input. In the domain of computer science, this necessarily means that there are no side effects; no print statements, saving to disk, and no use of global variables. If a function's computation is based in any way on a global variable, then it is possible for the same function, given the same arguments, to produce different results depending on the value of the global variable at the time it was run. This is in general a very bad way to write programs, and can be very confusing.

If we make our model out of more than one pure function (e.g. $f_1, f_2, ..., f_n$), then we could potentially chain the output of one function to the input of the next:

$$\begin{aligned} f_1(x_1) &\rightarrow x_2 \\ f_2(x_2) &\rightarrow x_3 \\ &\vdots \\ f_n(x_n) &\rightarrow x_{n+1} \end{aligned}$$

And we will always get the same output (x_{n+1}) at the end of the computational chain. In a perhaps more familiar notation we could have written

$$x_4 = f_3(f_2(f_1(x_1))) \tag{2}$$

All of the above has assumed that every function has no parameters. If we had a function which added 23 to it's input argument and then divided by three ($f_1(x_1) = (f_1 + 23)/3$), then we would be required to define a completely new function f_1 if we wanted to try adding 25 or dividing by a different number. If

some part of the function's behavior will be changing regularly and many values must be tried, then it makes sense to bundle these changing parts into a model parameter vector $\phi_1 = [\phi_{11}\phi_{12}$.

Now each function should look like this:

$$f_1(x_1, \phi_1) \rightarrow x_2 \tag{3}$$

And we chain them together as before:

$$\begin{aligned} f_1(x_1, \phi_1) &\rightarrow x_2 \\ f_2(x_2, \phi_2) &\rightarrow x_3 \\ &\vdots \\ f_n(x_n, \phi_n) &\rightarrow x_{n+1} \end{aligned}$$

Now we can describe nearly any neural model by specifying some initial conditions x_1, a list of functions $f_1, f_2, ..., f_n$, and a list of parameter vectors $\phi_1, \phi_2, ..., \phi_n$. By executing the functions one after another, our prediction about the neural activity will finally be held in x_{n+1}. If we want, we can observe intermediate values of x to better understand how the model is functioning as a whole.

The bundling of f and ϕ together is tentatively being called a **module**, for lack of a better way to describe each block of modeling functionality.

2 Module Technical Details

MATLAB's data types prohibit us from being completely general with regard to how we allowing functions f to do almost anything and how we use data x to represent almost anything. The best that we can do is to choose MATLAB's cell arrays in both cases, because of their generality and default behavior of reference-passing.

The structure which holds the list of functions $f_1, f_2, ...$ is therefore called `STACK` in weak analogy to the traditional computer data structure, and the structure which holds all the $x_1, x_2, ..., x_{n+1}$ is called `XXX` in weak analogy to the fact that each x is dependent upon the previous. Each cell of `STACK` and `XXX` is a structure containing named fields. You may put any fields you like in each cell, except for the fields defined in the next two sections, which have special meaning.

2.1 STACK data structure

A cell array containing the functions and their parameters that were applied to reach this point.

`STACK.name`	Function file name
`STACK.fn`	The function handle
`STACK.pretty_name`	User-readable pretty function name
`STACK.plot_fns`	Struct array with fields `pretty_name` and `fn`
`STACK.editable_fields`	Fields that may be user edited via the GUI.
`STACK.isready_pred`	A predicate function that is passed (STACK, X) and returns true iff it's ready to run.
`STACK.gh`	The "Gui Handles" structure, to be described in TODO.

2.2 Module Fields

If **F()** is a function to be used in the NARF architecture, it must satisfy a few core requirements to be used properly. These features are:

1. **F()** must be placed in a directory where NARF can find it.
2. **F()** must return a default struct **m** when given zero arguments. The structure **m** must satisfy the following requirements:
   (a) It *must* have these required fields:
   m.pretty_name An alphanumeric string describing the function
   m.fn A handle to a function which does the 'core computation' of **F().** It accepts two arguments (**p**, **x**). **p** is a copy of the **m** structure so that the state of **F()** may be easily read. **x** is a data structure passed from a previous function.
   m.input_validation_fn A function which accepts a two arguments (**p**, **x**), and returns a true if **m.fn(p,x)** can be run, otherwise it returns an error string.
   m.editable_fields A cell array of field names. These will be displayed to the GUI user for easy editing.
   (b) It *may* also have these optional fields which enable special behavior:
   m.plot_fns A cell array of cell arrays with three elements each. The first element in each sub-cell-array is a user-readable string naming the plot function, the second is a function handle to a function which plots on the default plot axes, and the third is a struct which contains parameters that will be passed to the plot function.
   (c) Any fields listed in **m.editable_fields** must have values with one of the following types: integer, floating point, 1D or 2D matrix, string, boolean, or cell array of strings. No other data types are allowed.
   (d) The user is allowed to define any other field in **m** that they wish and store arbitrary data in it. However, if it is not listed in **m.editable_fields**, it will not ever be directly editable.
3. When **F()** is given a single argument args, it must: (Footnote: a useful function to accomplish these two things is update_narf_struct.m)

(a) Check in args for the existence of fields listed in .

(b) Any fields that exist in args will be copied into m m to overwrite the default values of m.

2.3 XXX data structure

A cell array, with the most recent data being first. The contents of each cell could be anything. The following are typical values being used, but these are ONLY USED BY CONVENTION.

SYMBOL	DESCRIPTION	TYPE	SET OR MOI
`X.dat.().cellid`	Name of the cellid	String	-
`X.dat.().stimfile`	Name of the stimfile	String	-
`X.dat.().include_prestim`	Boolean. 1 prestim was included, 0 otherwise	Boolean	load_stim_resp
`X.dat.().raw_stim_fs`	Raw stimulus frequency	Double	load_stim_resp
`X.dat.().raw_resp_fs`	Raw response frequency	Double	load_stim_resp
`X.dat.().raw_stim`	Raw stimulus	[SxN]	load_stim_resp
`X.dat.().raw_stim_time`	Time vector for stimulus	[1xN]	load_stim_resp
`X.dat.().raw_resp`	Raw spike timings	[SxMxR]	load_stim_resp
`X.dat.().raw_resp_time`	Time vector for response	[1xM]	load_stim_resp
`X.dat.().raw_isi`	Raw inter-spike intervals		
`X.dat.().pp_stim`	Preprocessed stim		
`X.dat.().ds_stim`	Downsampled, preprocessed stim		
`X.dat.().ds_stim_time`	Time vector for downsampled stimulus		
`X.dat.().lf_stim`	Linear filtered stimulus (FIR or whatever)		
`.lf_preds`	Needs to be RENAMED		
`X.dat.().nl_stim`	Nonlinearly scaled stimulus		
`X.dat.().pred`	Prediction of the model		

In the above, dimensions are indicated with:

S = sound stimulus index #

R = repetition index #

N = Time index at the sampling rate of the stimulus.

M = Time index at the sampling rate of the response

T = Time index in downsampled frequency

F = Preprocessing index #

3 Overview of Typical Operations

The following is a rough overview of how typical operations affect the `STACK` and `XXX` structs.

Evaluation: Essentially, there is a chain of function calls, with the output of one function pushed onto the inputs of the next. Mathematically, it's easy to understand: `XXX{i+1} = STACK{i}.fn(XXX{i})`

Invalidation: If any intermediate parameter struct is modified, then it erases all `XXX` cells after it and the computation must recommence from that point.

Multiple models: If you need to do different 'branches' of computation or compare model structures, you can store the current computation `STACK` and save them somewhere, then load them back later.

Module loading: The only functions available are isted in the "modules" directory, which is read ONCE, at startup. They are only available from the popup selection when their ready_pred() function returns a true.

Editing: Fields in the `editable_fields` struct are GUI editable, but most other fields should be editable by passing the argument to the user-defined module creation funciton. However this is just a convention and relies on module authors to implement.

Graphing: Each module has (multiple) associated graphing functions which can be seleceted via a dropdown.

Error handling: Whenever you load or run a user-loadable function, a try-catch block around it to catch errors.

Saving and loading: When you want to save a model, just save the `STACK` data structure somewhere along with the GIT hash tag and initial data. Data from that point can always be reconstructed. When you want to load a model, loop through the `STACK` structure, starting from the first data X, and reconstruct the data as you go along.

Optimization pack/unpack: Packing operations go through `STACK` sequentially, pulling out any args found in the `fit_fields` cell array and putting them into a vector. The unpacking operation goes through `STACK` sequentially, pushing in any args with a FIT checkbox (accepts a vector as the input). TODO: During optimization, all controls should probably be disabled to avoid invalidation problems?

OPTIMIZATION There will need to be three types of modules: a performance metric, termination condition, and a sampling algorithm which picks out the next point to try. These are not part of the model explicitly, and must be reusable amongst different optimization algorithms.

4 Example Modules

TODO.

5 Example Script

See 'analysis/example_script.m' for more an example of how a linear model can be trained on an arbitrary data set.

6 Example Optimization

See TODO.org for more details. Clearly optimization is clearly NOT part of the model, yet we will want to plot various quantities and act in a modular way in a similar fashion to how the model already works. Pluggable sampling, termination conditions and objective functions are needed.

7 Further Development

See the TODO.org file in NARF's main directory.