PROGRAMMING TUTORIAL 8: BOLD RESPONSE

FABIAN SOTO

INTRODUCTION

- Do you remember the 3 steps to build and test a CCN model?
- Step 1: define units (e.g., LIF, AdEx)
- Step 2: define connectivity (e.g., synapse models, learning models)
- Step 3: relate activity to experimental measures (e.g., LFP model, decision-making model for choices and RTs)
- Last week, we learned a bit more about Step 3: we can use models of the BOLD response to transform “neural activity” values from a model into predictions for an fMRI study
- This week, you will learn how to actually do this in practice

GOAL OF THIS TUTORIAL

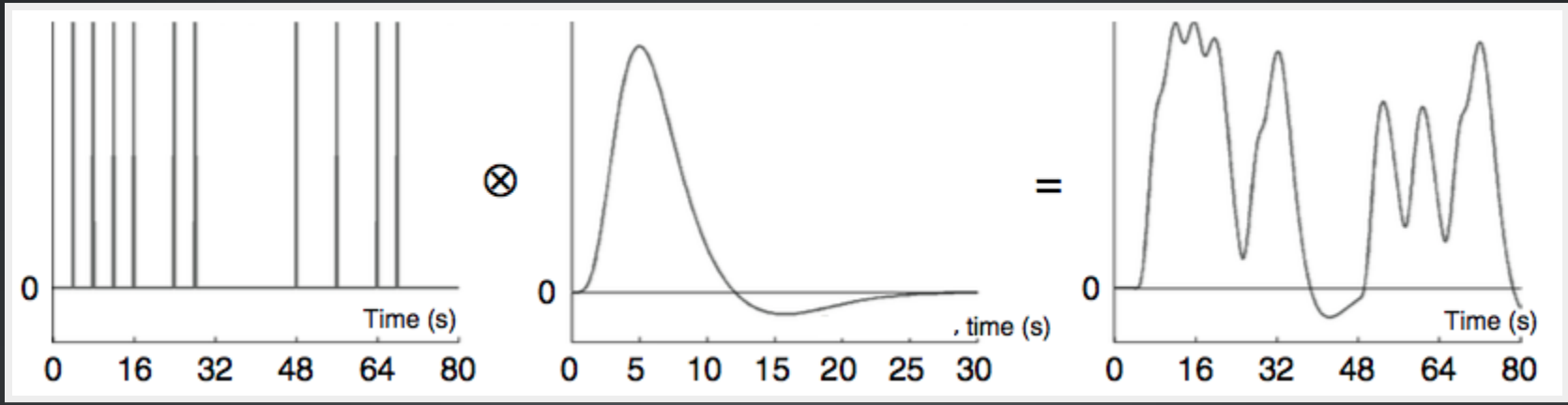
- In this tutorial, you will learn how to convert modeled “neural activity” values into predictions of BOLD response
- This is useful to performed model-based fMRI analysis
- For now, we will not use any specific model to generate the neural activity
- However, your last homework will involve converting reward prediction errors from our “critic” model to BOLD responses

BACKGROUND

- Let’s start by recapitulating what we learned:
- A model of the BOLD response takes as input a hypothetical “neural activity” value, which varies across time
- After applying some transformation on the input, it gives us back a BOLD response, which also varies across time
- There are two kinds of models: **linear** and **non-linear**

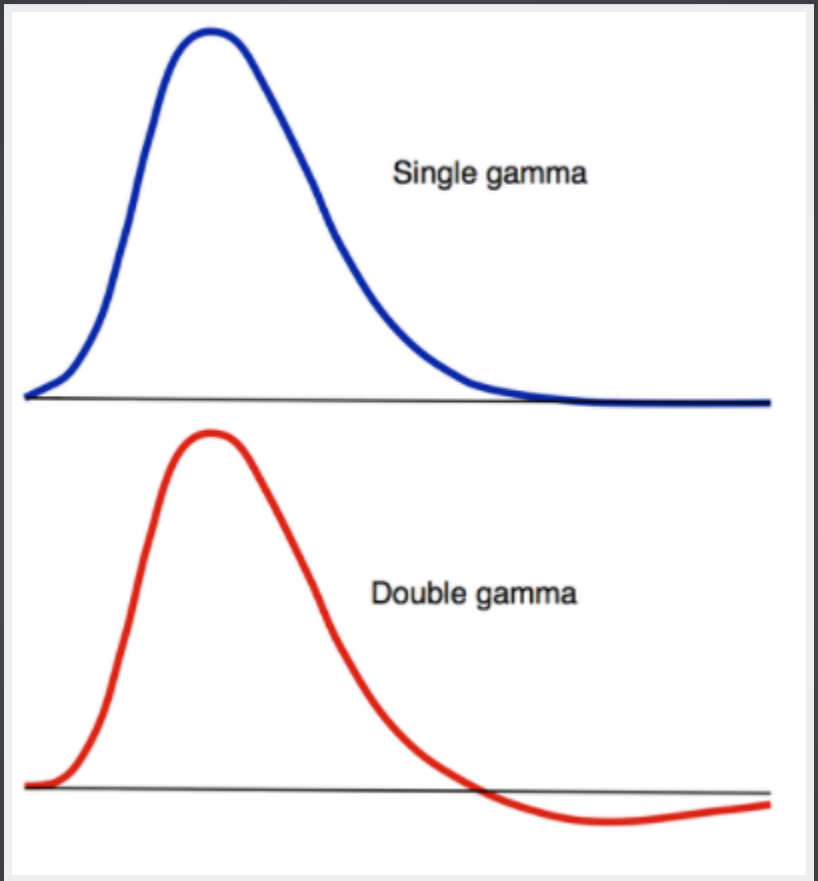
a) Linear models of BOLD

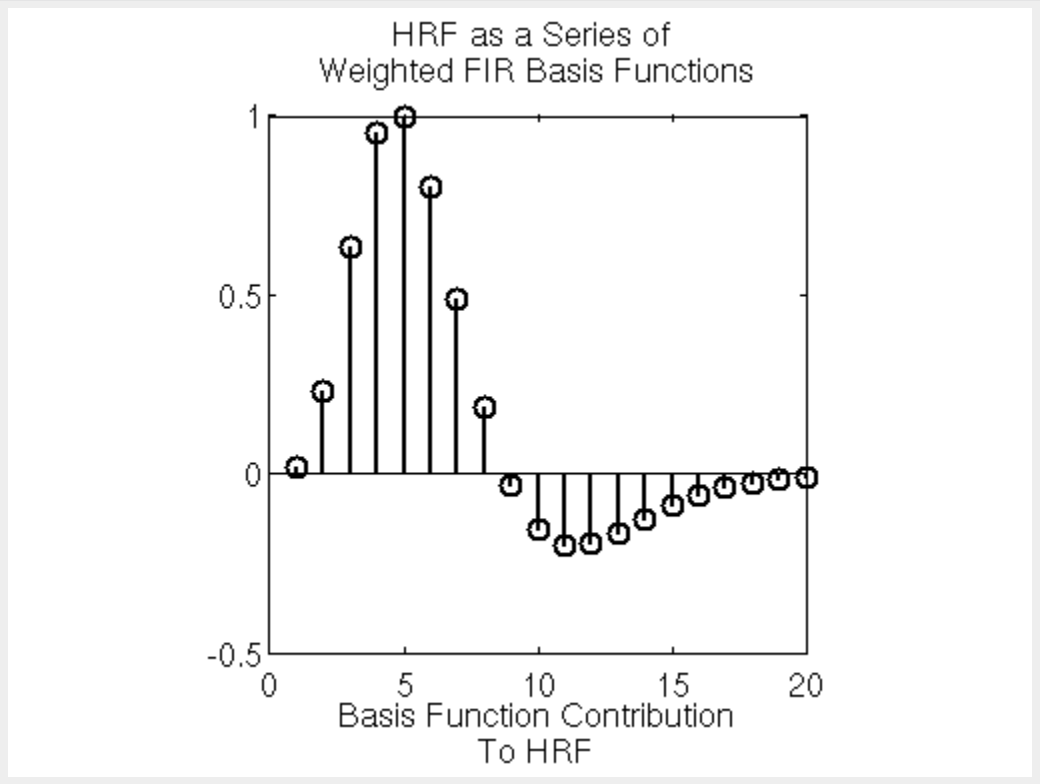
- These models assume that the BOLD response can be modeled as a linear system
- When a system is truly linear, all you need to know about it is its **impulse response function**, or IRF
- The system's response to any input is a convolution of that input and the IRF
- I'm sure that you know this stuff, after the trauma of having to solve this problem in the first quiz:



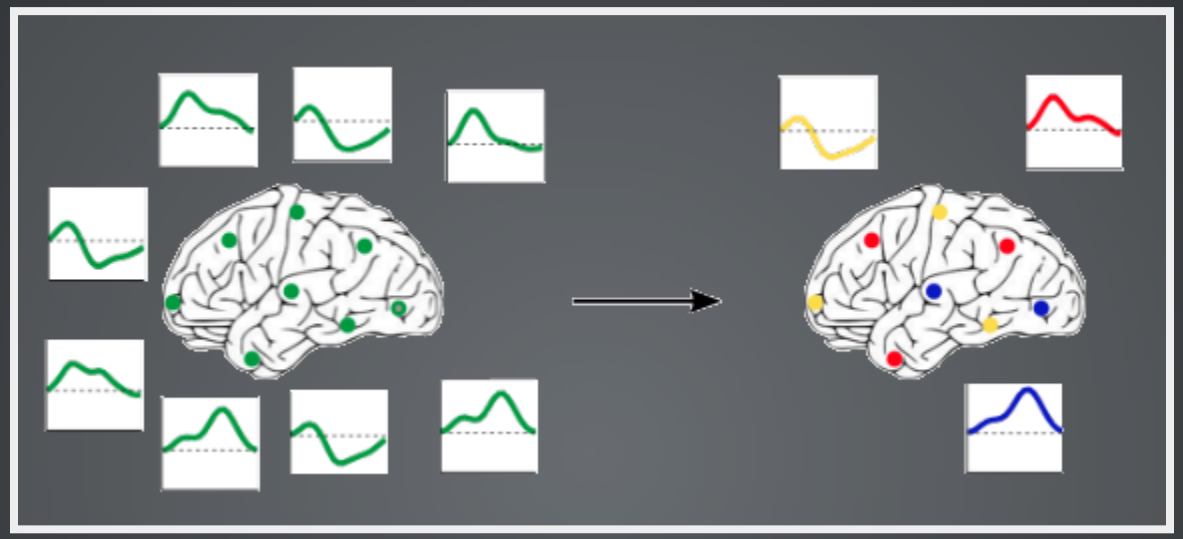
- In linear models of BOLD, the IRF becomes the HRF, or **hemodynamic response function**
- The HRF is the whole model, and neural activity timeseries simply get convolved with that HRF to obtain BOLD predictions

- There are many HRFs that can be used:




HRF as a Series of
Weighted FIR Basis Functions
1
0.5
0
-0.5
0
5
10
15
20
Basis Function Contribution
To HRF

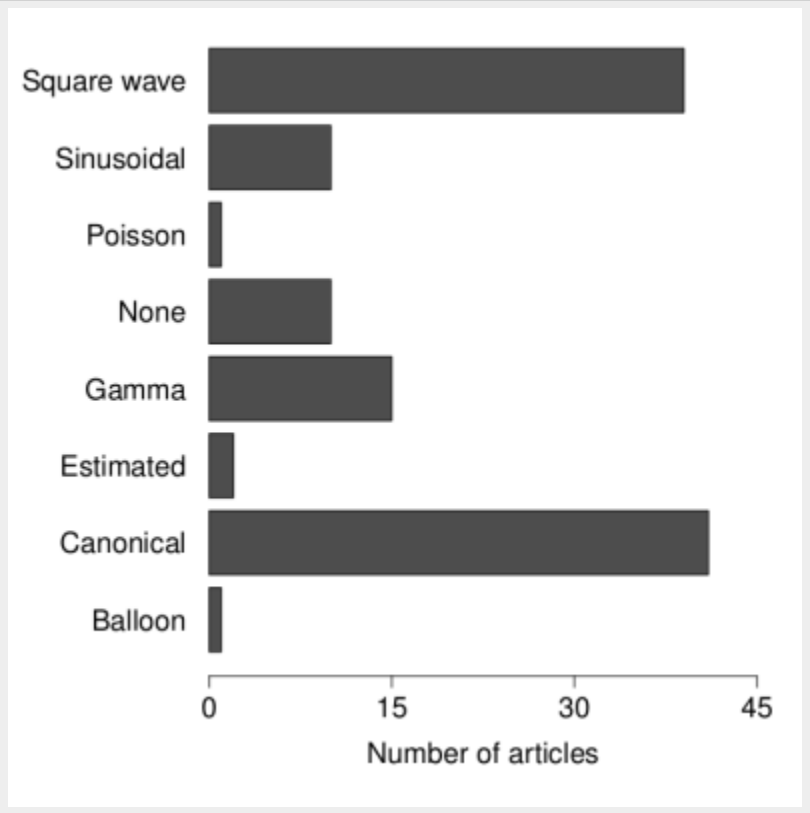
- An advantage of the FIR is that a different HRF can be estimated for each voxel:



b) Non-linear models of BOLD

- These models do not assume linearity
- The most famous example is the **balloon model**, which describes the BOLD as resulting from a nonlinear combination of blood volume and deoxyhemoglobin concentration
- These, in turn, are a nonlinear function of neural activity
- We also saw that a model using Volterra kernels can model some non-linearities in the BOLD signal, although it lacks the biological realism of the balloon model

- For better or worse, the linear model with a double-gamma HRF has become the standard in simulation work (Welvaert & Roseel, 2014):



USING THE LINEAR MODELS IN CCN

- In line with these results, here we will use the double-gamma HRF to get BOLD predictions
- CCN Models usually provide predictions of neural activity, which can be convolved with this HRF
- You can then search for voxels showing BOLD responses similar to the prediction
- This is called **model-based fMRI analysis**
- Let's go through each of the steps necessary to achieve this in python:

GETTING STARTED

- Open a jupyter notebook, and include some preliminary code in a new cell:

```
# import packages
import numpy as np
import matplotlib.pyplot as plt
from scipy.stats import gamma

# allow inline plotting in the notebook
%matplotlib inline
```

1. GET THE MODEL'S PREDICTIONS

- Let's manually create some arbitrary model predictions; this will prove to be helpful for your homework
- First, you need to understand what is the timescale of your model's predictions
- In most of our simulations we have divided time into steps of 1 ms
- However, for pedagogical purposes let's assume that our simulations have produced predictions in steps of 50 ms
- We will create predictions for a 2-minute scan, totaling 120 seconds, or 2,400 steps of 50 ms
- We start by creating a numpy array named `activity`, with only zeros:

```
# preallocate our "activity" array with zeros
activity = np.zeros(2400)

# create a "time" array in a scale of seconds, for reference
t = np.arange(.05,120.05,.05)
```

- The second line of code created a numpy array to keep track of the time, in seconds, corresponding to each time step in our `activity` array
- Imagine that the first stimulus presentation happens after a 5-sec waiting period, it lasts 500 ms, and it produces activity equal to 1.0
- Here's how we can write this first-trial activity on our `activity` array:

```
# the first stimulus is presented after 5 seconds, starting in step 101
# it lasts for 500 ms, or 10 steps of 50 ms
# it produces activity level 1.0
activity[(t>5) & (t<=5.5)] = 1
```

- Here, all we have done is choosing what elements of `activity` are equal to 1
- We do this by choosing time steps (in seconds, through the array `t`) that fulfill certain conditions
- They have to be larger than 5 seconds (`t>5`), because that is the initial waiting period
- Also (`&`), they have to be less or equal than 5.5 seconds (`t<=5.5`), because the stimulus lasts 500 ms (.5 seconds)

- Let’s check the result with a plot of only the first 6 seconds:

```
# we plot the early part (first 6 secs) of our activity array to check
plt.plot(t[0:120], activity[0:120])
plt.xlabel("Time (secs)")
```

- Note that 6 seconds = 120 steps of 50 ms (20 steps per second)

More trials

- Now let's assume that we want to have the same neural activity, equal to 1.0, in each of 14 trials
- As before, the experiment starts with a 5-second waiting period
- All trials look the same: activity for 500 ms, followed by an inter-trial interval (ITI) of 7 seconds
- How do we do this?
- One way is to loop through the trials using a `for` loop:

```
# we will loop trough 14 trials
for trial in range(1,15):
    # we need to calculate when to start presenting the stimulus
    stim_start = 5+(trial-1)*7.5

    # the stimulus end should be simply the start plus .5 secs
    stim_end = stim_start+.5

    activity[(t>stim_start) & (t<=stim_end)] = 1
```

- Here what we need to unpack is how we calculate the starting time for the stimulus in each trial
- The first stimulus is presented after 5 seconds, so that's our baseline
- Starting in the second trial (but not in the first one!), we want to add the duration of the previous trial (.5 secs) plus the ITI (7 secs) to the initial 5 secs

```
# we will loop trough 14 trials
for trial in range(1,15):
    # we need to calculate when to start presenting the stimulus
    stim_start = 5+(trial-1)*7.5

    # the stimulus end should be simply the start plus .5 secs
    stim_end = stim_start+.5

    activity[(t>stim_start) & (t<=stim_end)] = 1
```

- We can simply multiply `(trial-1)` by the sum of those values (`7.5`)
- Thus, the start of the stimulus will be 5 in trial 1 (when `trial-1` is zero), 12.5 in trial 2 (when `trial-1` is 1), 20 in trial 3, and so on
- The stimulus end is just the starting time plus 500 ms (the duration of the stimulus)

- After you run the `for` loop, take a look at how our “model predictions” look like with a plot:

```
plt.plot(t, activity)
plt.xlabel("Time (secs)")
```

2. CREATE A DISCRETE VERSION OF THE HRF

- The HRF is a continuous function, so we need to create a version of it in the discrete time steps that we have been using
- In our example, this is steps of 50 ms (or .05 secs)
- The HRF is back to zero in about 30 seconds, so we will create a series of time steps starting at .05 and going up to 30

```
# the HRF is back to zero after about 30 seconds
hrf_t =np.arange(.05,30.05,.05)
```

- After we obtain our discrete time steps in `hrf_t`, we simply evaluate two gamma functions (one with shape parameter equal to 6, the other equal to 12) at each time step

```
# we use a gamma to create the positive part of the HRF
positive_hrf = gamma.pdf(hrf_t, 6)

# we use a different gamma to create the negative part of the HRF
# this component has smaller amplitude, so we scale it down
negative_hrf = 0.3*gamma.pdf(hrf_t, 12)
```

- The first gamma (with shape parameter equal to 6) represents the initial increase in BOLD response
- The second gamma (with shape parameter equal to 12) represents the later undershoot
- Note that we have scaled the second gamma down, multiplying it by 0.3 (a in Ahsby's equation)

- Finally, we obtain the HRF by subtracting the second gamma from the first one:

```
# we substract one from the other to the HRF
HRF = positive_hrf - negative_hrf

# let's plot the HRF and its components
plt.plot(hrf_t, positive_hrf, label="positive part")
plt.plot(hrf_t, negative_hrf, label="negative part")
plt.plot(hrf_t, HRF, label="HRF")
plt.xlabel("Time (secs)")
plt.legend()
```

- The second part of this code creates a plot with the HRF and both of its components

3. CONVOLVE THE ACTIVITY AND HRF

- This part is super easy, we just enter `activity` and
hrf ~~`BOLD`~~ as inputs to numpy's convolution function
- The resulting convolved array will have more elements at the end than our original `activity` array
- We simply cut those out:

```
# convolution with numpy function
BOLD = np.convolve(activity, HRF)

# the convolved vector will have some extra points at the end
# we cut those out and set it to have the size of our activity vector
BOLD = BOLD[0:2400]

plt.plot(t, activity)
plt.plot(t, BOLD)
plt.xlabel("Time (secs)")
```

4. SUB-SAMPLING AND STORING

- Our actual fMRI experiment will not have this level of temporal precision
- The time steps in fMRI, represented by the TR, are usually around 1-2 seconds
- We need to **sub-sample** our BOLD timeseries; that is, we simply keep some of the values and drop all others
- Assuming a TR of 2 seconds, with the first sample at time t=2, we can keep all time steps that are multiples of 2
- We use the `np.mod(x, y)` function, which returns the remainder of dividing `x` by `y`. This remainder will be zero if `x` is a multiple of `y`:

```
# keep only time steps that are multiples of 2
# we are assuming TR = 2 secs
BOLD = BOLD[np.mod(t,2)==0]
t = t[np.mod(t,2)==0]

# Plot the predictions again
plt.plot(t, BOLD)
plt.xlabel("Time (secs)")
```

- To save our predicted BOLD to a text file, we use the following code:

```
# save a text file with our predictions
np.savetxt('predicted_BOLD.txt', BOLD)
```

- Now we can use the predictions as a regressor in our favorite fMRI data analysis software
- Hopefully, this is FSL, and hopefully you are using the randomise function to run permutation tests
- Remember: friends don’t let friends have a false-positive rate higher than 5%

ANOTHER EXAMPLE WITH CHANGING ACTIVITY

- Let's run the same code one more time, but this time the activity in each trial will be randomly sampled between 0 and 1 from a uniform distribution, using the function `np.random.uniform()`
- We start by creating the `activity` array:

```
# preallocate activity array and create a time array
activity = np.zeros(2400)
t = np.arange(.05,120.05,.05)

# we will loop trough 14 trials
# each trial has a stimulus presented for 500 ms
# followed by a 7-second ISI
for trial in range(1,15):
    # we need to calculate when to start presenting the stimulus
    stim_start = 5+(trial-1)*7.5

    # the stimulus end should be simply the start plus .5 secs
    stim_end = stim_start+.5

    activity[(t>stim_start) & (t<=stim_end)] = np.random.uniform()
```

- Now we run the convolution and plot the predictions (activity and BOLD) at high temporal precision (50 ms time steps):

```
# convolution with numpy function
BOLD = np.convolve(activity, HRF)

# the convolved vector will have some extra points at the end
# we cut those out and set it to have the size of our activity vector
BOLD = BOLD[0:2400]

plt.plot(t, activity)
plt.plot(t, BOLD)
plt.xlabel("Time (secs)")
```

- Finally, we subsample to TR = 2 secs, we plot the predicted BOLD and store to a file:

```
# keep only time steps that are multiples of 2
# we are assuming TR = 2 secs
BOLD = BOLD[np.mod(t,2)==0]
t = t[np.mod(t,2)==0]

# Plot the predictions again
plt.plot(t, BOLD)
plt.xlabel("Time (secs)")

# save a text file with our predictions
np.savetxt('predicted_BOLD.txt', BOLD)
```

IN CASE YOU KNOW R...

- For consistency, this tutorial was in Python, as the rest of the course
- However, you should know that there is another option: an R package called **neuRosim**
- Here's a link to the paper describing it: https://www.jstatsoft.org/article/view/v044i10
- It's easy to use, and includes the Balloon model
- Furthermore, it allows to add different kinds of noise to your modeled BOLD signals

THAT'S IT!