Programming Tutorial 6: Modeling a "Critic" and Dopamine Release

Fabian Soto

Goal of this tutorial

- In this tutorial, we will use Python to implement a simple model of reward learning
- This is a model that learns the reward value of different stimuli
- We will later apply this model to modify synapses in our decision making model
- That is, we want our model to learn reward values for two stimuli: “green” and “red”

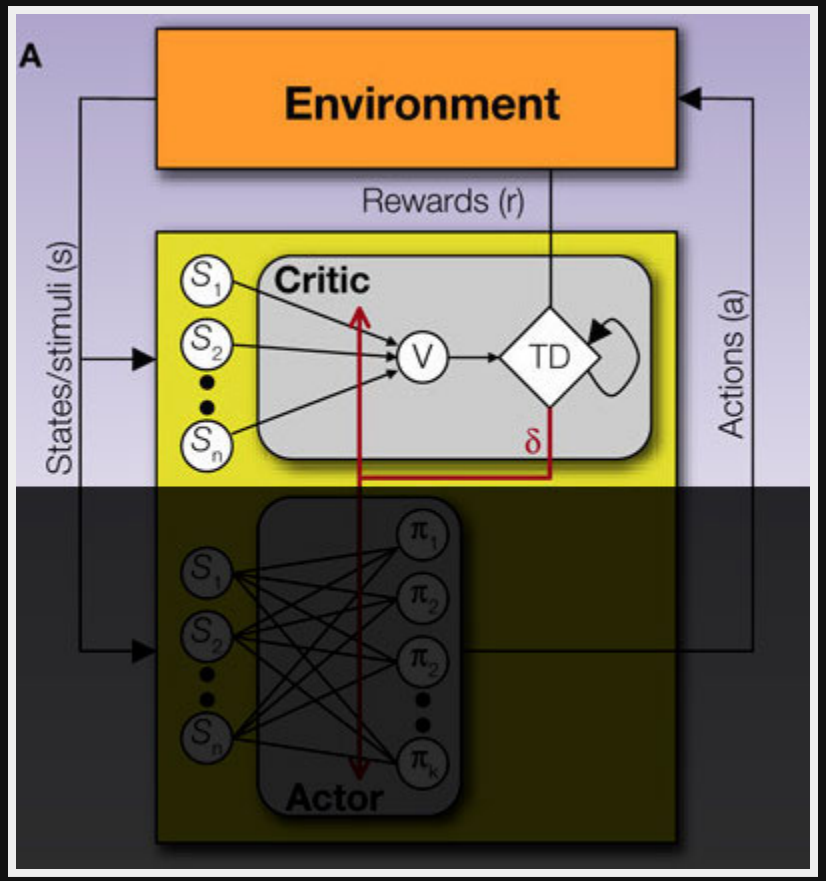
The RPE hypothesis of dopamine

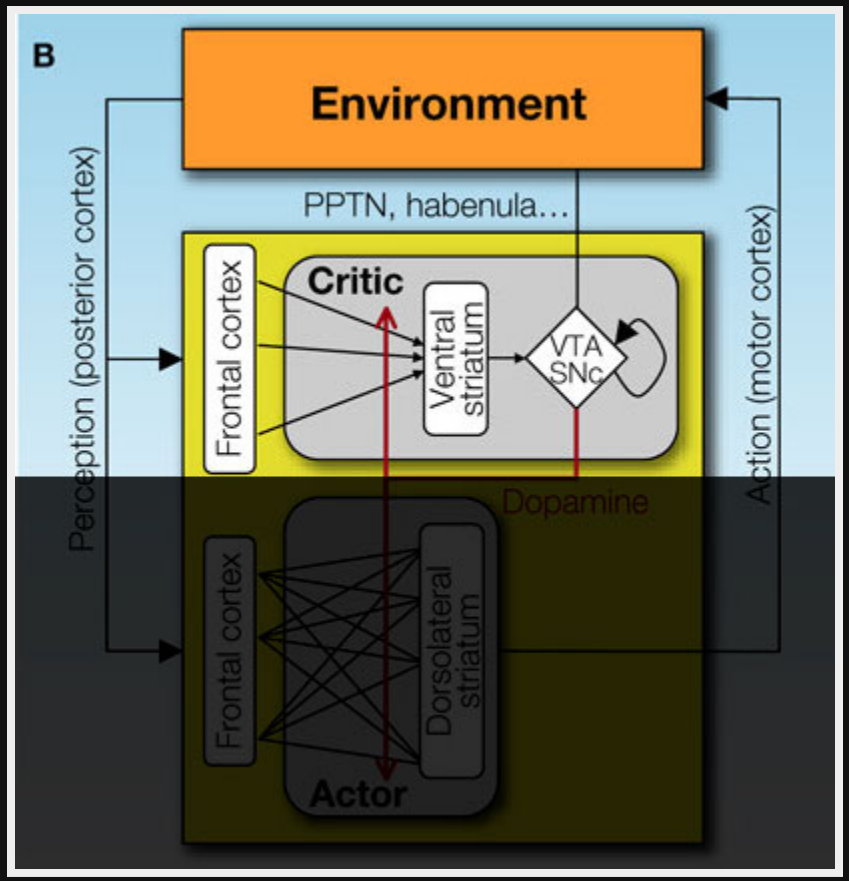
- The first step in building a CCN model is determining units and what they compute
- Up to this point, we have focused mostly on single neurons as units
- The allowed computations were very simple: summation, thresholding (action potential), adaptation, etc.
- More complex computations are possible when many neurons are arranged in a circuit (e.g., decision making circuit)
- Sometimes, it is possible to summarize the computations in a circuit through a couple of equations
- The equations will describe the output from the circuit, which usually corresponds with activity in a specific neuron or homogeneous group of neurons within the circuit
- We care only about what the circuit computes, not exactly how it computes it

- This is the case with reward prediction error (RPEs) as a model for firing of dopaminergic (DA) neurons
- DA activity is quantitatively predicted by the RPEs described by some models of Pavlovian conditioning (the Rescorla-Wagner model, TD learning).
- It is possible to create detailed models of how that RPE is computed in a circuit
- However, most applications don’t do this; they simply use the RPE equations directly as a model of firing rate of DA neurons
- This RPE is used to modulate learning about the value of stimuli and actions

The actor-critic model

- In the next couple of weeks we will be working with the actor-critic model of reinforcement learning
- We will start by working with the critic:


A
Environment
Rewards (r)
States/stimuli (s)
Actions (a)
Critic
V
TD
δ
Actor


B
Environment
PPTN, habenula…
Perception (posterior cortex)
Action (motor cortex)
Frontal cortex
Critic
Ventral striatum
VTA
SNc
Dopamine
Frontal cortex
Dorsolateral striatum
Actor

Getting started

- Open a jupyter notebook
- Include the standard lines to import tools for your simulation
- This time, we will not need Brian2

```
# import packages
import numpy as np
import pylab as plt

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

1. Computing RPEs

a) Keep track of each stimulus value and reward

- Let's start with an example with two stimuli: green and red
- First, create a numpy array with two values, representing whether a stimulus is presented (1) or not (0)
- We do this with `S = np.array([1, 1])`
- Second, we need an array with the reward value of each stimulus, represented by V_i
- We do this just as before; for example: `V = np.array([0.2, 0.1])`
- Finally, we can store whether a reward was presented (1) or not (0) in a variable named `rw`
- Now we have all the ingredients to compute our RPE

b) Compute the RPE

- We will use the simple definition of RPE from the Rescorla-Wagner model:
- $\delta = \lambda - \sum_i S_i V_i$
- This is pretty much the same definition as in the Niv paper, but we are using the variable S_i to indicate whether a stimulus is present or not, and thus whether its reward value is added to the global reward prediction
- The Python code that does this is: `RPE = rw-np.sum(S*V)`
- Note that we are using the sum function from numpy, because `S` and `V` are numpy arrays
- If you put all these steps in the same jupiter cell, you should have something like the following:

```
# we store an array representing whether a stimulus is presented or not
S = np.array([1, 1])

# we store an array with the current reward values of each stimulus
V = np.array([0.2, 0.1])

# we store the actual reward in a variable (0 or 1)
rw = 1

# we compute the RPE
RPE = rw-np.sum(S*V)

# print RPE
print RPE
```

- Here we added a last statement to print the RPE to screen
- Try computing the RPE when both stimuli are present, both have $V = 1$ and the reward is present. What happened?
- Now try $V_1 = 1$ and $V_2 = -1$ with the reward present. What happened?

2. Updating reward values

- In the Rescorla-Wagner model and TD learning, reward values V_i are updated in each trial, in proportion to the RPE:
- $\Delta V_i = S_i \eta \delta$
- Here, ΔV_i is the change in reward value of stimulus i
- η is a learning rate parameter, determines how fast or slow the reward values are learned
- Finally, new reward values are the old values plus their change: $V_i = V_i + \Delta V_i$
- All of this can be implemented by the following code:

```
# we set a learning rate
eta = 0.25

# the new reward values are updated according to RPE
delta_V = S*eta*RPE
V = V + delta_V

print V
```

- Copy this code to a new cell
- Re-run the two cells that you have with both stimuli and reward present, $V_1 = 0.1$ and $V_2 = 0.2$. What are the new reward values? Why?

3. Put everything in a function

- We need to apply these steps to update reward values in each learning trial, and a simulation will have many trials
- Thus, it is a good idea to place everything in a function
- We will want the function to return two things: the RPE and the new V_is
- To do this, we can use `return`:

```
def my_function(input1, input2, ...):
  output1 = we_do_stuff_to(input1)
  output2 = and_we_do_stuff_to(input2)

  return output1, output2

# now we call our function with different inputs
x, y = my_function(a, b)
x, y = my_function(c, d)
```

- Let’s apply this general syntax to our example
- In a new cell, paste the following code:

```
# the input to our function should be S, the current V, reward value, and eta
def update_critic(S, V, rw, eta):
    RPE = rw-np.sum(S*V)
    delta_V = S*eta*RPE
    V = V + delta_V

    return RPE, V

# we test it with the values for S, V, rw and eta that we already had
RPE,V = update_critic(S, V, rw, eta)
print RPE
print V
```

- The first part creates the function
- The second part tests that the function is working, using the example that we completed before
- You should get the same results as before when you print `RPE` and `V`

4. Simulate simple learning and extinction

- Let's use our function to simulate how the reward value of a single stimulus changes during learning and extinction
- We will simulate 20 learning trials, in which the stimulus is present (`S = 1`) and so is the reward (`rw = 1`)
- This will be followed by 20 extinction trials, in which the stimulus is present (`S = 1`) but the reward is not (`rw = 0`)
- We start by creating 2 numpy arrays, with the values of `S` and `rw` across all 40 trials
- We will use only three numpy functions: `np.zeros`, which creates arrays with zeros, `np.ones`, which creates arrays with ones, and `np.append(A, B)` which appends the array `B` at the end of array `A`:

```
# create an array indicating that a stimulus is presented in 40 trials
S = np.ones(40)

# create an array with reward values across trials
rw = np.append(np.ones(20), np.zeros(20))

# check that both arrays are correct
print S
print rw
```

- When you print both arrays at the end of this code, you should see 40 ones in array `S`
- You should see 20 ones followed by 20 zeros in array `rw`

- What we want now is to use our function `update_critic` to update the reward value of our stimulus
- To do this, we will create a `for` loop to cycle through all 40 values in `S` and `rw`
- In each iteration of the loop, we will compute `RPE` for that trial and update the reward value `V`, which will result in two new arrays
- In this kind of situation, it is a good practice to create the full array `RPE` and `V` before running the for loop, placing zeros (or other values) in each element
- We will create 41 zeros for `V`, the initial value of zero plus the updated values after each trial: `V = np.zeros(41)`
- We will create 40 zeros for `RPE`: `RPE = np.zeros(40)`

- Finally, we pick a value for the learning rate (`eta=0.25`) and loop through all trials, updating `V` and `RPE`:

```
# preallocate numpy arrays for V and RPE with zeros
# V will have an extra value at the end
V = np.zeros(41)
RPE = np.zeros(40)

# pick learning rate
eta = 0.25

# loop through all trials, updating reward values
for trial in range(0,40):
    # update critic
    RPE[trial], V[trial+1] = update_critic(S[trial], V[trial], rw[trial], eta)
```

- Now we can plot the reward value across all trials with the following code:

```
# plot the reward value across trials
plt.plot(V)
plt.xlabel("Trials")
plt.ylabel("Reward Value (V)")
plt.ylim(-0.1,1.1)
```

5. Simulate the blocking effect

- The blocking effect involves two stimuli: A and B
- First, A is paired with a reward (or any other US) repeatedly (20 trials here)
- Then, the compound AB is paired with the same reward (20 additional trials)
- At the end of this training, B is tested and it usually shows no evidence of being associated with the reward
- That is, because A is a perfect predictor of the reward, this “blocks” B from acquiring any reward value

- Blocking involves two stimuli, so now each trial involves arrays `S` and `V` with two values each
- We will create matrices to deal with them: columns will represent stimuli (A and B) and rows will represent trials
- Let's start with `S`: it is composed of two columns, one representing presentations of A and one representing presentations of B
- We will create two separate arrays `S_A` and `S_B`, using the same strategy as before: a combination of `np.zeros`, `np.ones` and `np.append`
- However, now we will make sure to create arrays with rows and columns, by giving inputs `(row_number, col_number)` to `np.ones` and `np.zeros`
- Furthermore, we will tell `np.append` exactly how we want to paste together different arrays setting `axis=0` (columns side by side; `axis=1` would concatenate them in a single column):

```
# create an array with reward values across trials
rw = np.ones(40)

# create an array indicating that stimulus A is presented in 40 trials
S_A = np.ones((40,1))

# create an array indicating that stimulus B is NOT presented in the first 20 trials,
# and it is presented in the last 20 trials
S_B = np.zeros((20,1))
S_B = np.append(S_B, np.ones((20,1)), axis=0)

# put the two arrays together in a matrix, with rows representing trials and columns st
S = np.append(S_A, S_B, axis=1)
```

- Here the last line places the two columns `S_A` and `S_B` side by side, creating a 40x2 matrix `S`
- The first line creates `rw` as before
- You should check that the arrays look correct by using `print S` and `print rw`

- Now we are ready to run our simulation using the `update_critic` function
- We will use the same `for` loop as before, but now we need to create a 40x2 matrix of zeros when we preallocate `V`
- Also, in each iteration of the `for` loop we will now use whole rows from the `S` and `V` matrices: `S[trial,]` and `V[trial,]`
- Create a new cell and copy the following code, which runs the simulation and plots the results:

```
# pick learning rate
eta = 0.25

# preallocate numpy arrays for V and RPE with zeros
# V will have an extra value at the end
V = np.zeros((41,2))
RPE = np.zeros(40)

# loop through all trials, updating reward values
for trial in range(0,40):
    # update critic
    RPE[trial], V[trial+1,] = update_critic(S[trial,], V[trial,], rw[trial], eta)


# plot the reward value across trials
plt.plot(V)
plt.xlabel("Trials")
plt.ylabel("Reward Value (V)")
plt.ylim(-0.1,1.1)
```

TD learning

- Here we used the RW model to learn reward values
- This is good if all you care about is predictions of RPE in each trial
- If you care about the RPE in a milisecond scale, it's better to use TD learning
- This is necessary when timing is important (e.g., to explain effects of delaying reward)
- This is also necessary when you want to explain measurements of neural activity that are precise in time, as in single-neuron recordings and LFPs
- This is not necessary when you measurements are more coarse, as with fMRI

That's it!