PROGRAMMING TUTORIAL 7: REINFORCEMENT LEARNING

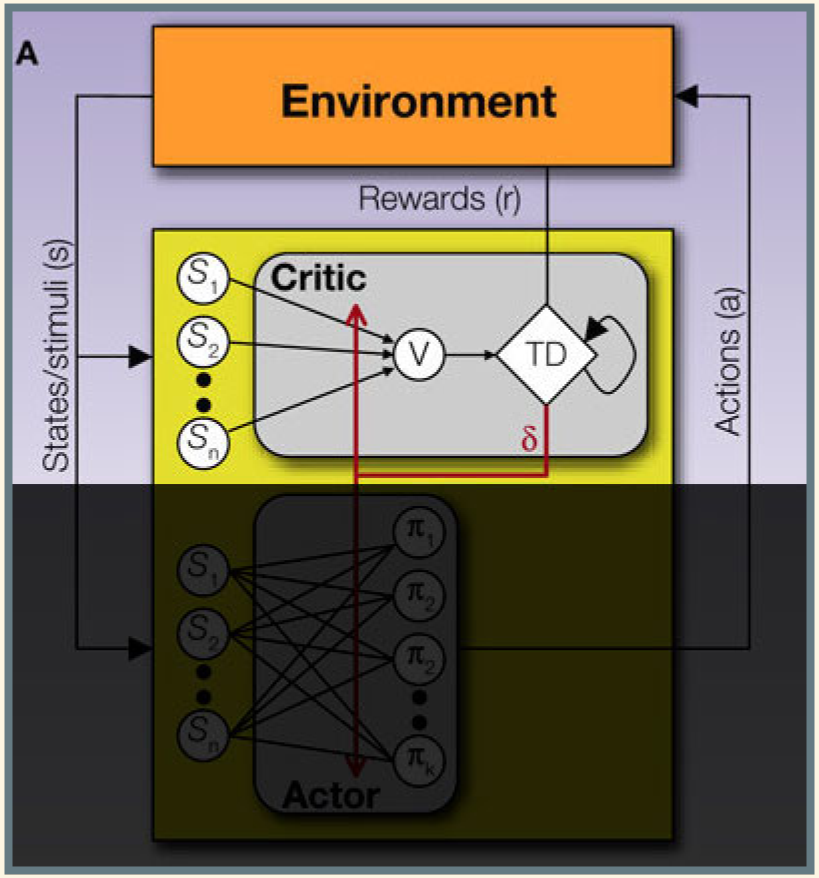
FABIAN SOTO

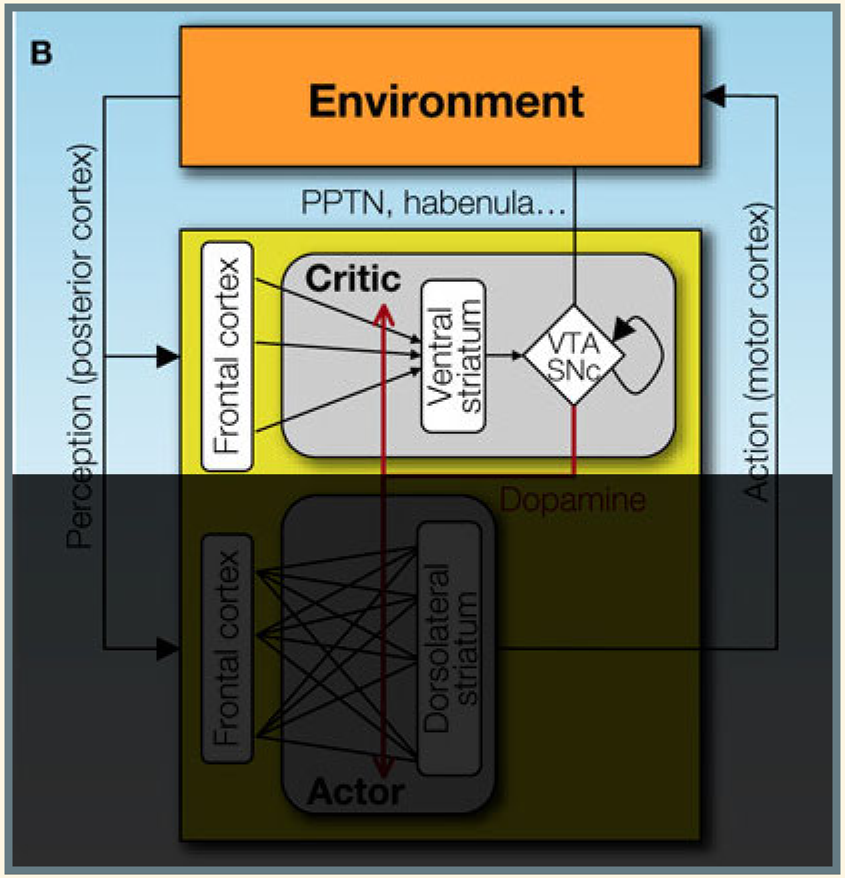
GOAL OF THIS TUTORIAL

- In this tutorial, we will implement a full reinforcement learning model using python and Brian2
- The reinforcement learning model will use an “actor-critic” architecture
- We will use two models that we already implemented earlier
- The decision-making network will be our “actor”
- The reward learning model will be our “critic”
- Our model will learn to give different responses to “green” and “red” stimuli

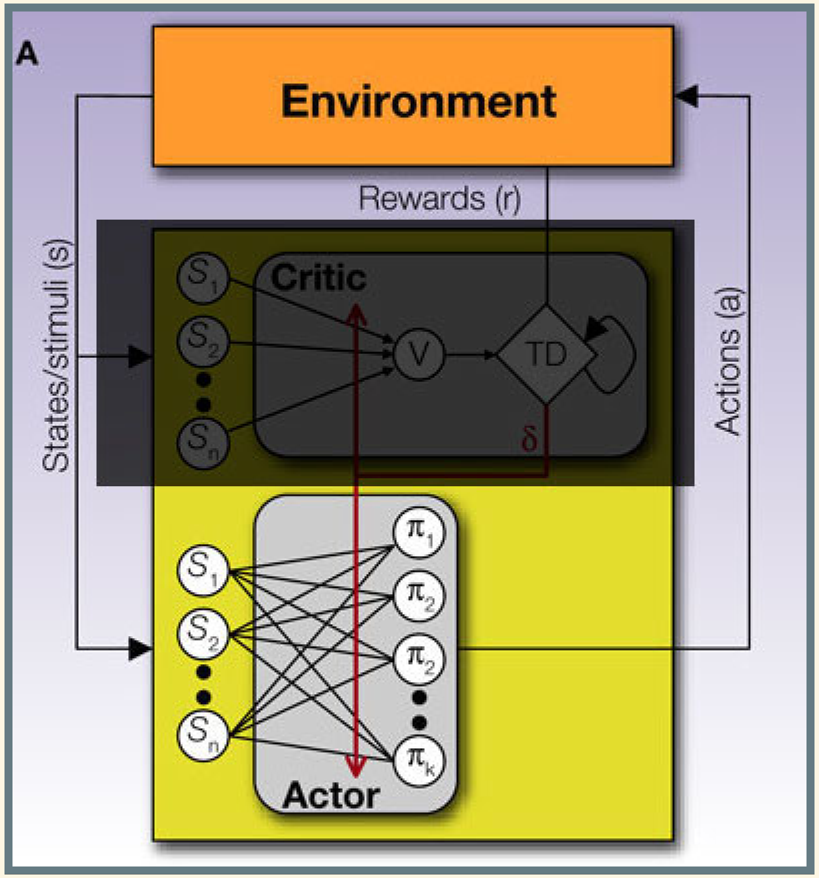
A REVIEW OF THE ACTOR-CRITIC MODEL

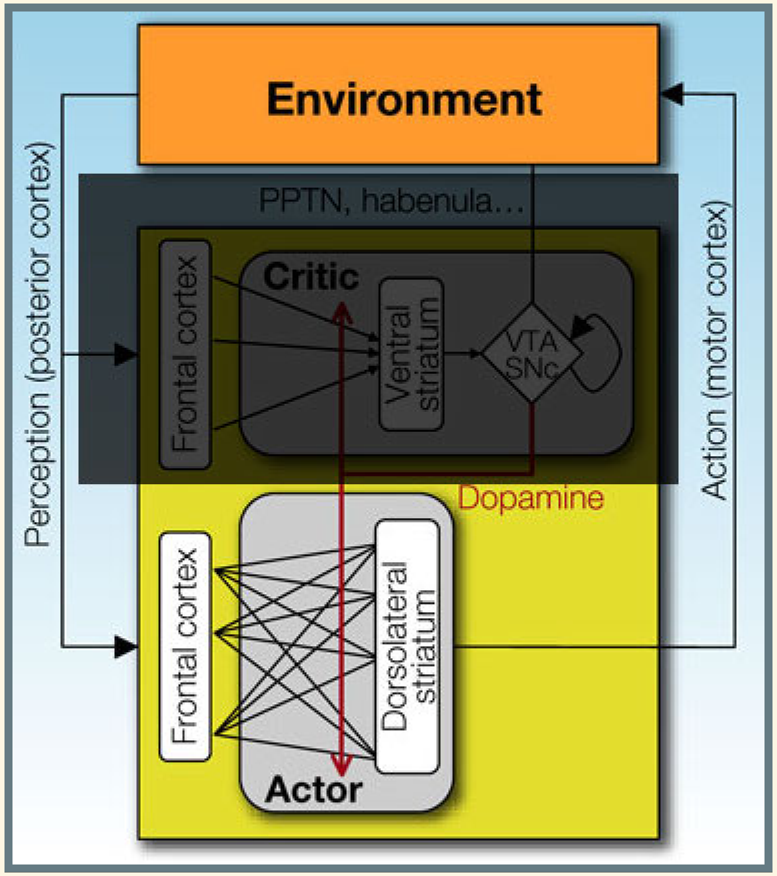
The critic




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

The actor


A
Environment
Rewards (r)
States/stimuli (s)
Actions (a)
Critic
S_1
S_2
S_n
V
TD
δ
π_1
π_2
π_2
π_k
Actor


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

GETTING STARTED

- Open a jupyter notebook
- Include the standard lines to import tools for your simulation:

```
# import packages
from brian2 import *
import numpy as np
import pylab as plt

# allow inline plotting in the notebook
%matplotlib inline

# start scope for this brian2 simulation
start_scope()
```

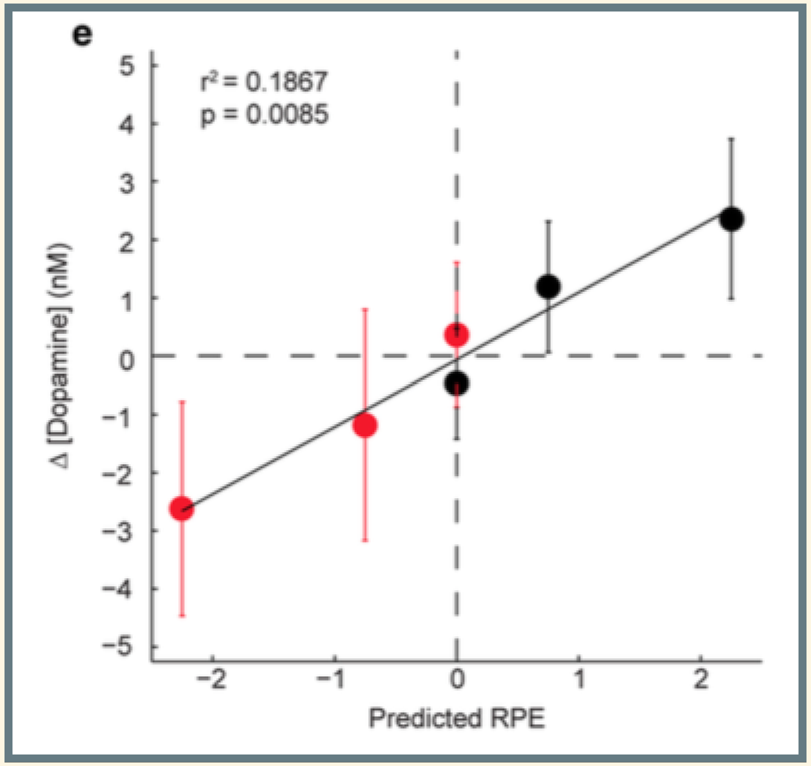
1. DEFINE THE CRITIC

- In programming tutorial #6, we created a function that learns reward values for stimuli and computes RPEs
- We called that function `update_critic`
- We will use this function to simulate a critic that learns reward values for each stimulus
- Copy and paste the function definition in a new cell:

```
# define the function to update critic (from Programming Tutorial #6)
# the input to our function should be S, the current V, reward value, an
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

# Start array V at zero
V = np.array([0, 0])

# Set learning rate
eta = 0.2
```

- Note that we have also started the array `V` (reward values) at zero
- We also set the learning rate for reward values `eta` to 0.2

- We will assume that dopamine discharge in the striatum is proportional to the RPE computed from our critic model
- There is evidence suggesting that this assumption is correct (Hart et al., 2014):



2. DEFINE THE ACTOR

- We will use our simple decision making network as the actor
- We have already programmed this in Programming Tutorial #5 and Homework #4, so you should be familiar with the model
- We will essentially copy-paste cells that create this model in Brian2:

2.1 Neuron and synapse equations

```
# define equations for the AdEx model (include noise)
# and equations for the AMPA and GABA synapse models
eqs =  '''
du/dt = ( -(u - u_rest) + delta_T*exp((u-vartheta_rh)/delta_T) +
        R*(I - w - I_AMPA - I_GABA) ) / tau_m +
        sigma*xi*tau_m**-0.5 : volt
dw/dt = (a*(u - u_rest) - w)/tau_w : amp
I = input_current(t,i) : amp
I_AMPA = g_AMPA*(u - E_AMPA): amp
I_GABA = g_GABA*(u - E_GABA): amp
dg_AMPA/dt = -g_AMPA / tau_AMPA : siemens
dg_GABA/dt = -g_GABA / tau_GABA : siemens
'''
```

- Remember that the equation for du/dt should be in a single line!

2.2 Define parameters of the synaptic models

```
# AMPA synapses
E_AMPA = 0 * mV                        # equilibrium potential of the channe
tau_AMPA = 2 * ms                      # speed of NT unbinding
AMPA_jump = 200*nsiemens               # jump in conductivity for excitatory

# GABA synapses
E_GABA = -70 * mV              # equilibrium potential of the channel
tau_GABA = 6 * ms              # speed of NT unbinding
GABA_jump = 80*nsiemens        # jump in conductivity for inhibitory synapse
```

2.3 RSN model

```
# create neuron group
RSN = NeuronGroup(2, eqs, threshold='u>theta_reset',
      reset='u=u_r; w+= b')

# define parameters of the RSN AdEx model
RSN.namespace['R'] = 200*Mohm                 # membrane resistance
RSN.namespace['tau_m'] = 24*ms                # membrane time constant
RSN.namespace['theta_reset'] = 35*mV          # reset threshold
RSN.namespace['u_rest'] = -65*mV              # resting potential
RSN.namespace['u_r'] = -55*mV                 # reset potential
RSN.namespace['vartheta_rh'] = -52*mV         # rheobase threshold
RSN.namespace['delta_T'] = 0.8*mV             # sharpness of the action potent
RSN.namespace['tau_w'] = 88*ms                # adaptation time constant
RSN.namespace['a'] = -0.8*nS                  # subthreshold adaptation consta
RSN.namespace['b'] = 65*pamp                  # adaptation jump after a spike
RSN.namespace['sigma'] = 0*mV                 # amplitude of noise added to tl
```

- continues…

```
# set initial potential to u_rest
# use the variable in this model's namespace (see below)
RSN.u[:] = -65*mV

# set initial adaptation to zero
RSN.w[:] = 0

# Manual input to this model (zeros for now)
input_current = np.zeros((2000,2))
input_current = TimedArray(input_current*pamp, dt=1*ms)
RSN.namespace['input_current'] = input_current

# let's get rid of input_current to avoid namespace warnings
del input_current
```

2.4 MSN model

```
# create neuron group
MSN = NeuronGroup(2, eqs, threshold='u>theta_reset',
      reset='u=u_r; w+= b')

# define parameters of the RSN AdEx model
MSN.namespace['R'] = 31.5*Mohm              # membrane resistance
MSN.namespace['tau_m'] = 50*ms              # membrane time constant
MSN.namespace['theta_reset'] = 40*mV        # reset threshold
MSN.namespace['u_rest'] = -80*mV            # resting potential
MSN.namespace['u_r'] = -55*mV               # reset potential
MSN.namespace['vartheta_rh'] = -55*mV       # rheobase threshold
MSN.namespace['delta_T'] = 2*mV             # sharpness of the action potent
MSN.namespace['tau_w'] = 100*ms             # adaptation time constant
MSN.namespace['a'] = -20*nS                 # subthreshold adaptation consta
MSN.namespace['b'] = 150*pamp               # adaptation jump after a spike
MSN.namespace['sigma'] = 5*mV               # amplitude of noise added to tl
```

- continues...

```python
# set initial potential to u_rest
# use the variable in this model's namespace (see below)
MSN.u[:] = -80*mV

# set initial adaptation to zero
MSN.w[:] = 0

# Manual input to this model is all zeros
input_current = np.zeros((2000,4))
input_current = TimedArray(input_current*pamp, dt=1*ms)
MSN.namespace['input_current'] = input_current

# let's get rid of input_current to avoid namespace warnings
del input_current
```

2.5 Connect the neurons

- This is the only part of the model that we will modify
- Previously, we defined different values for the jump in AMPA current (excitatory) after a presynaptic spike
- We had `AMPA_jump_large` representing strong excitatory connections, and `AMPA_jump_small` representing weak excitatory connections
- Here, these connections will be learned, so we will define only one value for the jump in AMPA, called `AMPA_jump`
- This value will be multiplied by a synaptic weight or efficacy, represented by the variable `W` (capital letter)

```
# create excitatory synapses
E = Synapses(RSN, MSN, 'W : 1', pre='g_AMPA += AMPA_jump*W')
E.connect(0,0)
E.connect(1,1)
E.connect(0,1)
E.connect(1,0)

# create inhibitory synapses
I = Synapses(MSN, pre='g_GABA += GABA_jump')
I.connect(0,1)
I.connect(1,0)
```

2.6 Set up the monitor to record spikes

```
# indicate what to record
spikes_RSN = SpikeMonitor(RSN)
spikes_MSN = SpikeMonitor(MSN)

# store the model
store()
```

3. RUN A SINGLE TRIAL

- Create a new cell
- The first thing that we need to define in a trial is what stimulus has been presented
- In our case, we have two stimuli: “green”, represented by the index 0, and “red”, represented by the index 1
- We will create a variable called `presented_stimulus` that has this information for the current trial
- When a stimulus is presented, we will use the value in `presented_stimulus` to determine which RSN should receive a manual input of 400pA:

```python
# variable determines what stimulus is presented
# 0=green, 1=red
presented_stimulus = 1

# set up manual input to RSNs (changes in "green" vs. "red" trials)
# see PT05 and HW4 for details
trial_input = np.zeros((2000,2))
trial_input[:,presented_stimulus] = 400
trial_input = TimedArray(trial_input*pamp, dt=1*ms)
RSN.namespace['input_current'] = trial_input
```

- Next, we will restore the decision model to its saved configuration using `restore()`
- We then set the value of `W` to anything we want
- Finally, we run the simulation and calculate `RT` and `choice` as in our previous simulations
- Add the following at the bottom of the cell that you last created:

```
restore()

# give a value to the excitatory synaptic weights
W = np.array([0.3, 0.3, 0.3, 0.3])
E.W[:] = W

# run the simulation for 2 s
run(2000*ms)

# calculate RT and choice
RT = 100 + 1000*np.amin([spikes_MSN.t[spikes_MSN.i==0][4],
      spikes_MSN.t[spikes_MSN.i==1][4]])
choice = np.argmin([spikes_MSN.t[spikes_MSN.i==0][4],
      spikes_MSN.t[spikes_MSN.i==1][4]])
```

3.2 Determine whether a reward is given and compute RPE

- Because the reward depends on whether the model chooses the correct response, we need to determine this
- In our case, if the index stored in `presented_stimulus` (0=green or 1=red) is the same as the index stored in `choice`, then we give a reward
- To check this, we can use an `if, then` statement in python
- We tell python that if `choice==presented_stimulus`, then reward should be 1, otherwise it should be zero
- In a new cell, include the following:

```python
# determine whether a reward is given
if (choice==presented_stimulus):
    rw = 1
else:
    rw = 0
```

- Now that we have the obtained reward, we can update the reward values `V` using our `update_critic` function
- For this, we first create an array `S`, representing what stimulus was presented during the trial
- Include the following in your cell:

```python
# Create the array S
S = np.zeros(2)
S[presented_stimulus] = 1


RPE,V = update_critic(S, V, rw, eta)
```

3.3 Using RPE, update weights of the decision network

- In a typical model-free RL model, the weights between states (i.e., the RSNs representing “green” and “red”) and actions (i.e., the MSNs representing responses) are updated using three pieces of information: the activity representing states, activity representing actions, and RPE.
- The weights in our decision network are stored in `W`, in the following order:

- `w[0]` = RSN_0 to MSN_0 (“green” stimulus and “green” action)
- `w[1]` = RSN_1 to MSN_1 (“red” stimulus and “red” action)
- `w[2]` = RSN_0 to MSN_1 (“green” stimulus and “red” action)
- `w[3]` = RSN_1 to MSN_0 (“red” stimulus and “green” action)
- We will create corresponding arrays representing “state” and “action” representations (RSN and MSN activity levels, respectively)

- First, we create the state representation. In our case, we will use the firing rate of each RSN as the representation of “green” and “red”.
- The spiking of each of these neurons is stored in `spikes_RSN.count[0]` for “green” and `spikes_RSN.count[1]` for “red”
- To keep weights within bounds, we will divide this activity by 120, which is around the maximum number of spikes that we can see in this network
- This will give us the value of the variable “RSN activity”, represented by `RSN_A`:

```
# Compute activity variables for RSNs
RSN_A = np.zeros(4)
RSN_A[[0,2]] = spikes_RSN.count[0]/double(120)
RSN_A[[1,3]] = spikes_RSN.count[1]/double(120)
```

- Second, we create the action representation. Again, we will use the transformed firing rate from MSNs as our representation of the responses “green” and “red”:

```
# Compute activity variables for MSNs
MSN_A = np.zeros(4)
MSN_A[[0,3]] = spikes_MSN.count[0]/double(50)
MSN_A[[1,2]] = spikes_MSN.count[1]/double(50)
```

- Finally, we will update the weights in `W` according to the following rule:
- $W_{ij} = W_{ij} + \alpha A_{MSN_i} A_{RSN_j} \delta$
- α is a learning rate
- A_{MSN_i} is the activity in MSN i, A_{RSN_j} is the activity in RSN j
- δ is the RPE

```
# apply update rule
alpha = 0.3
W = W+alpha*RSN_A*MSN_A*RPE
```

A FUNCTION TO PRINT RESULTS TO SCREEN

- In Homework #5, you will have to run many of these trials and keep track of what is going on with your network
- To check what is happening as the simulation runs, it is convenient to print the results from each trial to the screen
- This allows you to understand how learning develops, and also lets you know whether there is some issue or unexpected result in your simulations as they run
- I have created a small function that allows you to print to screen a summary of a trial
- Create a new cell and copy-paste the following:

```python
def print_trial_info(trial, presented_stimulus, choice, rw, RPE, V, W):
    stim_string = ['green','red']
    print 'Trial #'+str(trial)
    print 'Presented stimulus: '+stim_string[presented_stimulus]
    print 'Choice: '+stim_string[choice]
    print 'Reward: '+str(rw)
    print 'RPE: ' +str(RPE)
    print 'Updated reward values:'+str(V)
    print 'Updated synaptic weights: '+str(W)+'\n'

print_trial_info(1, presented_stimulus, choice, rw, RPE, V, W)
```

RUNNING SEVERAL TRIALS

- Running several trials involves simply including all the steps from point 3 of this tutorial within a `for` loop
- You will have to also initialize the variables `V` and `W` before the start of the loop, and update them in each cycle
- Finally, you should include a call to `print_trial_info` at the end of the loop
- Let me show you more or less how this should look like

THAT'S IT!