Assignment 2
Neuroscience of Learning, Memory, Cognition
Dr.Aghajan

Mohammad Mowlavi

February 29, 2024

Contents

Chapter 1

Assignment 2

1.1 Generalized Firing-Rate-Based Network Models

1.1.1 Explain the Code and Results

As mentioned in the exercise, we use odeint to solve the differential equation. We give this function 4 inputs, differential equation function, initial value, time steps and finally pars values. And this function iteratively solves the equation.
The function which I passed to the odeint is $network_equations$. This function return the equations below which is provided in the notebook:

$$\frac{dI_s}{dt} = \frac{-I_s + w.u}{\tau_s}$$
$$\frac{dv}{dt} = \frac{-v + F(I_s(t))}{\tau_r}$$
$$F(x) = \frac{1}{1 + e^{-x}}$$

Here is the code:

```
# The function you pass to odeint as an argument. you must calculate dv/dt at each timestep
def F(x):
return (1 / (1 + np.exp(-x)))

def network_equations(y0, time, pars):
# ================================================================
dIsdt = (-y0[0] + np.dot(pars['weights'], pars['firing_rates'])) / pars['tau_s']
dvdt = (-y0[1] + F(y0[0])) / pars['tau_r']
return [dIsdt, dvdt]
# ================================================================

def run_network():
# Initialize the output firing rate and total input current with zero
initials = [0, 0]

# Set up the timescale
time = np.linspace(0, 20, 1000)

#solve the ODE using odeint
pars = def_params()
sol = odeint(network_equations, initials, time, args=(pars,))

```

```
# Generate the demanded figure and plots
# ================================================================

fig, (ax1, ax2) = plt.subplots(1, 2, figsize=(16, 6))
ax1.plot(time, sol[:, 0], linewidth=4)
ax1.set_xlabel('Time')
ax1.set_ylabel('Total Input Current (Is)')
ax1.set_title('Total Input Current')

ax2.plot(time, sol[:, 1], color='red', linewidth=4)
ax2.set_xlabel('Time')
ax2.set_ylabel('Postsynaptic Neuron Activity (v)')
ax2.set_title('Postsynaptic Neuron Activity')

plt.tight_layout()
plt.show()
# ================================================================
```

The results of running code is shown in figure 1.1 and the results are as expected.

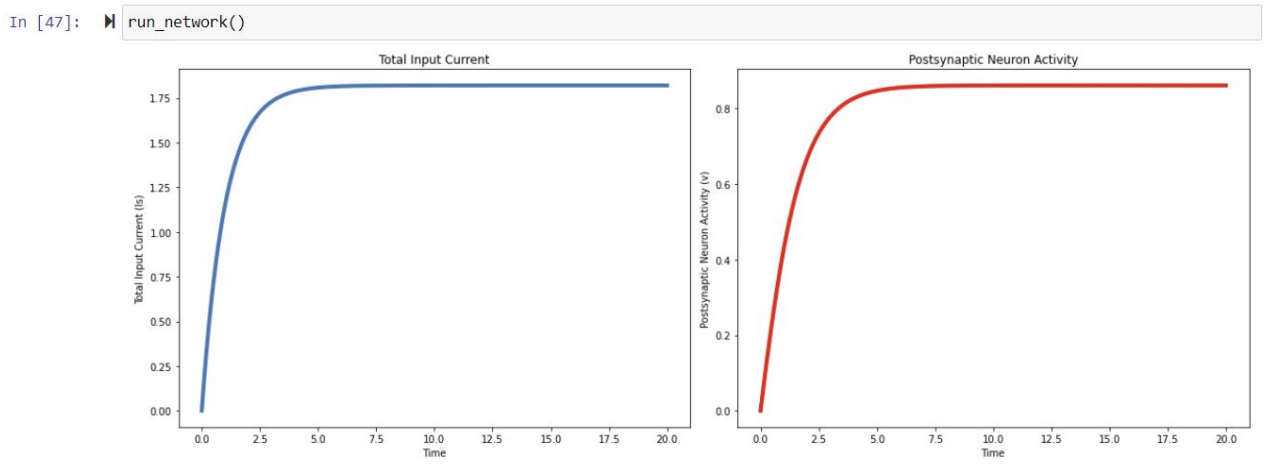

Figure 1.1: Result of simulation.

1.1.2 Make the Input Reach a Steady State Much Faster Than the Output

This is possible because reaching the steady state depends on the time constants and if we change them, we can naturally satisfy the request of the question. There are two ways to do this. Either we can decrease the time constant of the input current or we can increase the time constant of the output. In both of these cases, the fast input of the thesis reaches the steady state from the output.
First I change the time constant of inputs (τ_s). The results are shown in figure 1.2.
Now, I change the time constants of output (τ_r). The results are shown in figure 1.3.
In both figures 1.2, 1.3 inputs reach to the steady state faster than output.

1.1.3 Try out the Model for Other Non-Linear Functions

As we know, there are several other non-linear activation functions such as Hyperbolic Tangent, Leaky ReLU, ReLU, Exponential Linear Unit and ...
We need to choose the activation function that can model the behavior of the neurons well. We also know that the behavior of neurons is such that the more the current increases, the longer their voltage increases, and of course this has a limit, that is, it does not continue indefinitely due to some physical reasons and etc.
The sigmoid function handles this case well, and because this layer is derivable, it makes the process of gradient descent and learning easier. Another function that behaves like this function is the tanh function and I will use

In [47]:
```
run_network()
```

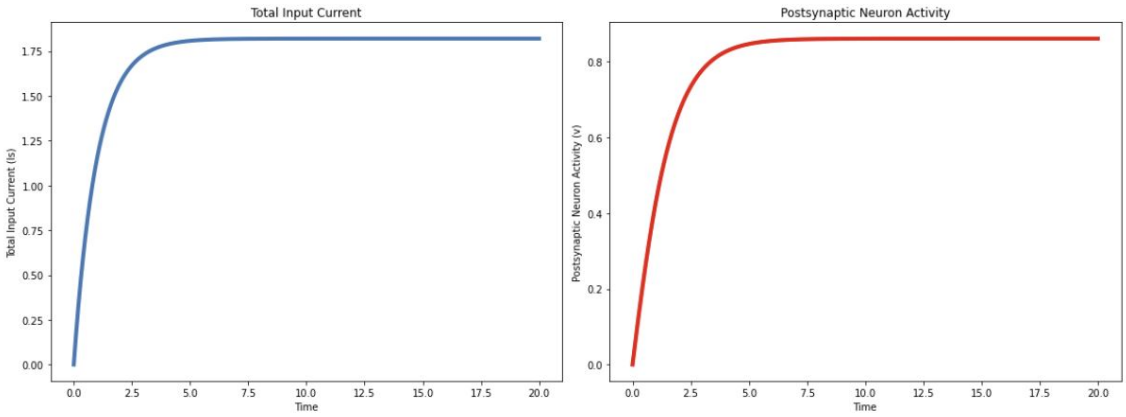

Figure 1.2: $\tau_s = 0.5$ and $\tau_r = 1$

In [47]:
```
run_network()
```

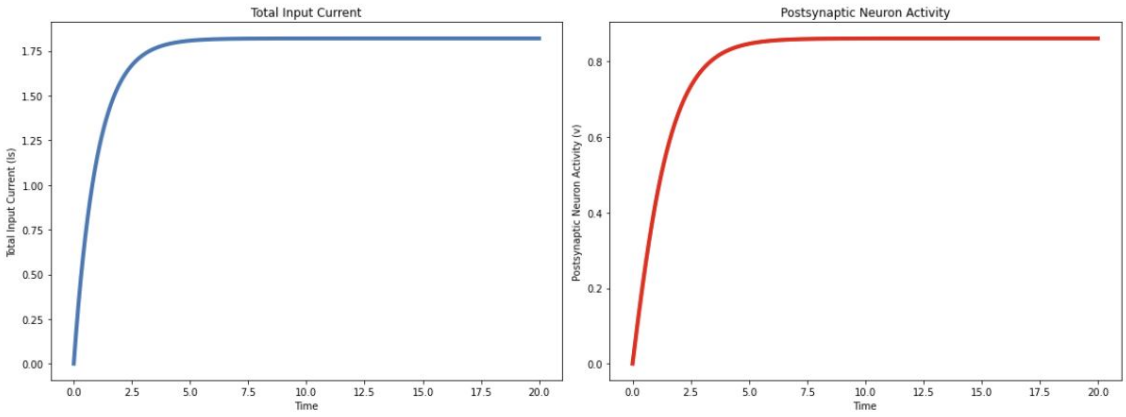

Figure 1.3: $\tau_s = 1$ and $\tau_r = 2$

it. The result is shown in figure 1.4. As you can see, both the final value in the steady state of the voltage has increased and it has reached the steady state later. As a result, it has differences with the previous activator function.

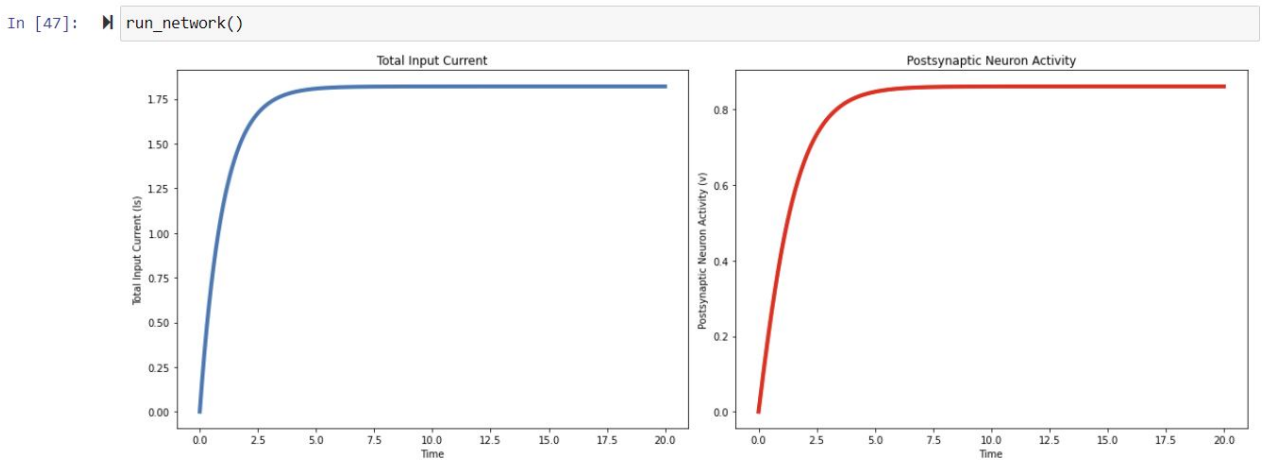

Figure 1.4: using tanh instead of sigmoid activation function.

1.1.4 Compare This Model With a Perceptron Network

I think there are differences between these two models. For example, the activation function of perceptron model returns only zero and one and is discrete, while the activation function of this model returns continuous results between zero and one, and this helps us more for better simulation. Also, the same difference can be seen in the output. The perceptron model only has an output of one or zero, while this model has an output between zero and one and is continuous, just as the output voltage and input current of neurons are continuous. Also, the learning process is different in these two models. As you can see in the code, we used derivation and gradient descent methods for learning, while in the perceptron model we use some other perceptron method. In fact, the perceptron learning rule is a form of supervised learning and is suitable for binary classification tasks.
I think we cannot classify the input current as well as the previous model. Because in the previous model, the input current was continuous and the output voltage was continuous, but in the perceptron model, the input current is continuous but output is not continuous and this is a challenge.

1.2 Reservoir Computing

This section does not need a special explanation and all the code items written in Jupiter Notebook are specified. I just want to say that for the first part we reached 100% accuracy as expected and for the second part 71% accuracy, which of course can be different with each run because the outputs are randomly manipulated.