Matteo Zortea, Alessandro Rizzi, Marvin Wolf

G3A8 - Exercise sheet 5

Exercise 1

(a)

When we plot the output firing rate (figure 1) as a function of the constant input current (bias) for $T_{sim}/\tau_m \approx 20$ we notice that there are some "stairs" and the function is not continuos as predicted by our model

$$\nu(I) = \frac{1}{\tau_{\mathrm{ref}} + \tau_{\mathrm{m}} \log\left(\frac{E_{\mathrm{reset}} - u_{\mathrm{eff}}}{\Theta - u_{\mathrm{eff}}}\right)}$$

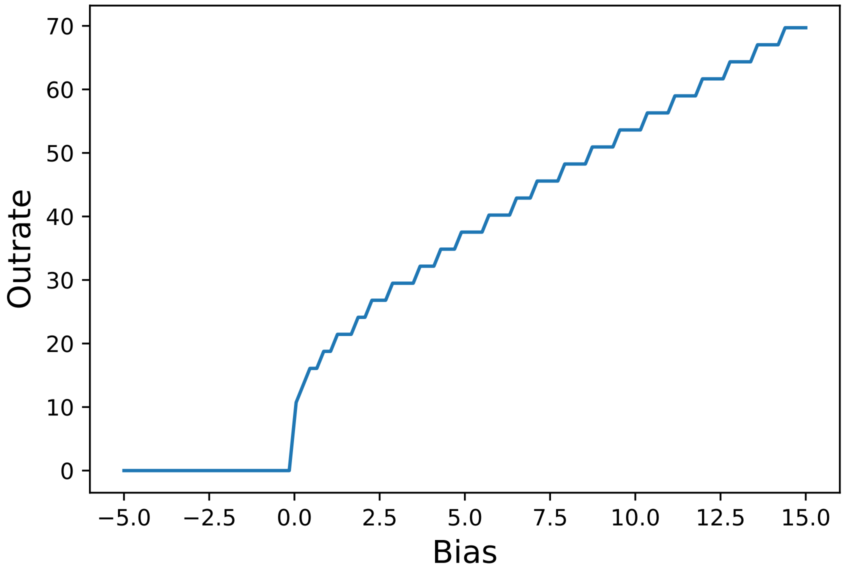


Figure 1: $\tau_m = 20$ ms, $T_{sim} = 373$ ms

The effect decreases for higher simulation times (figure 2) and increases for lower simulation times (figure 3).

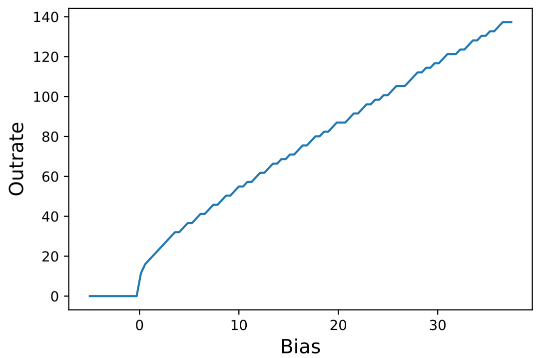


Figure 2: $\tau_m = 20$ ms, $T_{sim} = 437$ ms

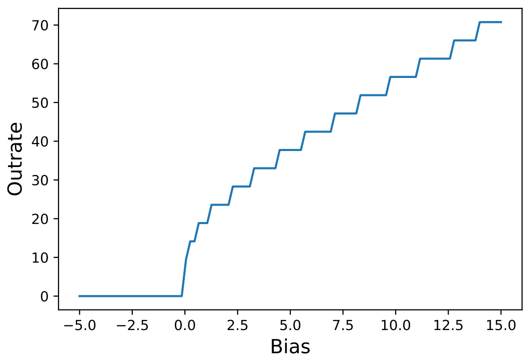


Figure 3: $\tau_m = 20$ ms, $T_{sim} = 212$ ms

The behaviour disappears in the limit of infinite simultion time (figure 4) and holds

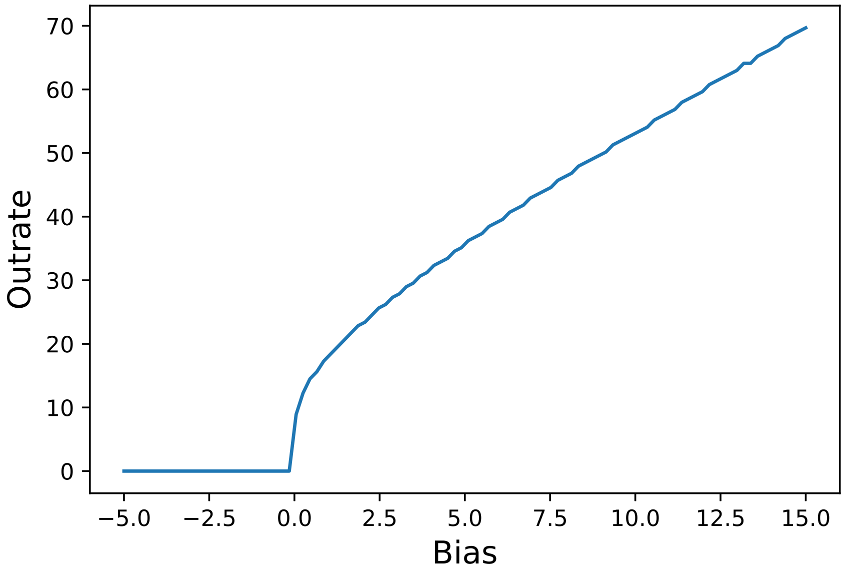


Figure 4: $\tau_m = 20$ ms, $T_{sim} = 1794$ ms

indepentently of the value of τ_m (figures 5 and 6).

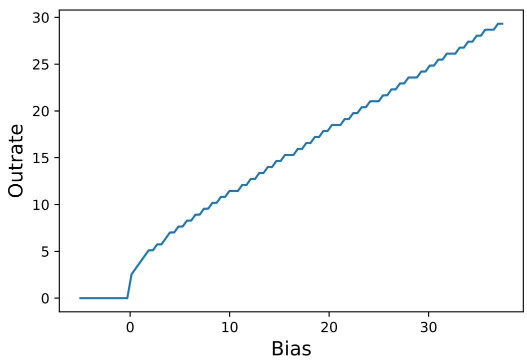


Figure 5: $\tau_m = 100$ ms, $T_{sim} = 1569$ ms

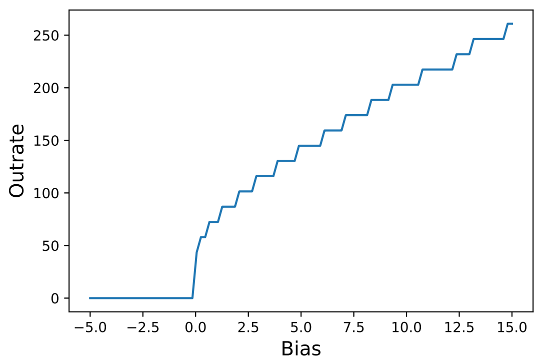


Figure 6: $\tau_m = 5$ ms, $T_{sim} = 69$ ms

We conclude that this is just a mere effect of how we calculate the spiking rate. Indeed, this is calculated as the number of spikes divided by the total simulation time. If the simulation time is not long enough compared to τ_m, one finds that a small variation (ideally infinitesimal) in the input current does not result in small variation of the spiking rate (ideally infinitesimal) because there can be either one spike more or zero. The result is always guaranteed to be correct in the limit $T_{sim} \to +\infty$. If T_{sim} is not long enough (typically one wants $T_{sim}/\tau_m \gg 100$), when we increase the bias over the value that gives us one more spike, we significantly affect the calculation of ν (N or $N+1$ has a significant difference on ν if N is small).

In the limit $\tau \to 0$ one gets a constant spiking rate indepentently of the input curret as shown in figure 7. Normally u_{eff}, which contains the input current, determines the equilibrium value for u, and the further u from u_{eff}, the faster the change of u in time.

Hence, the value of the input current obviously affect the spiking rate since the latter depends on how fast u reaches the threshold point. If $\tau_m \to 0$, the output instantaneously follows the input indepentently of the values of the latter. The spiking rate, then, cannot depend on the input in this case, provided that the input is such that $u_{eff} > u_{thresh}$, otherwised no spikes are observed.

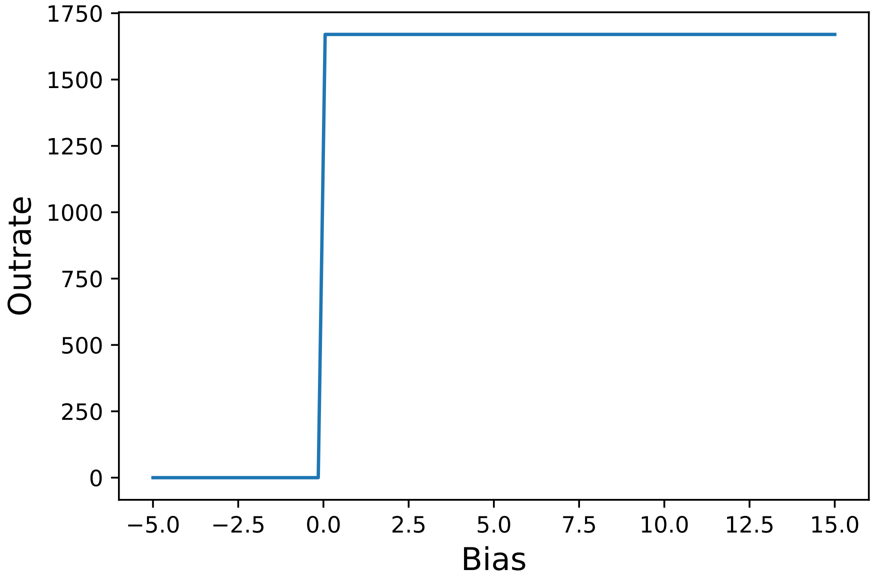


Figure 7: $\tau_m = 10^{-50}$ ms, $T_{sim} = 100$ ms

b)

NOTE When we made the plots, we set the values of the quantities manually instead of using the sliders. Unfortunately we noticed only after exporting the plots, that the pre-built function does not use the bias values used in the simulation, but the ones set in the sliders, which, in our case, were not those used in the simulation.
Since we always kept the bias constant at -3.0, and the sliders are normally set to -5.0 mV, the corrected plots just differ from those shown by a shift upwards of 2 mV of u_{eff}. This does not change the meaning of the comments we made. We apologise for the problem.

For constant weight (figures 8-11), one has that a higher input rate results in a higher average value and maximum value of u, after the initial transient effect is gone. This is because every input spike pushes the output higher and the latter decays with time constant τ_m, hence the higher the frequency, the higher the number of "pushes" to u, the less time for u to discharge. If the input rate is high enough so that the output has not enough time to discharge and - on the opposite - charge even more, we can eventually reach the spike threshold.

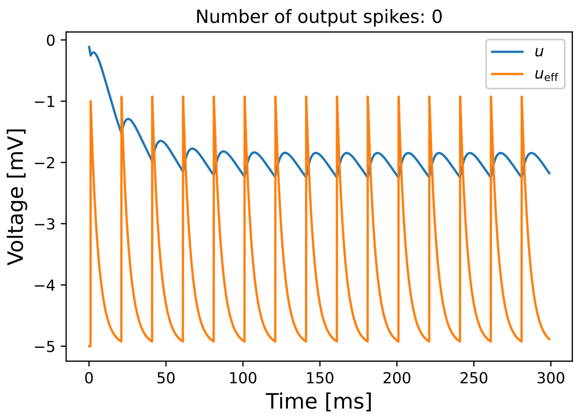


Figure 8: Weight 50, bias -3.0 mV, rate 50 Hz

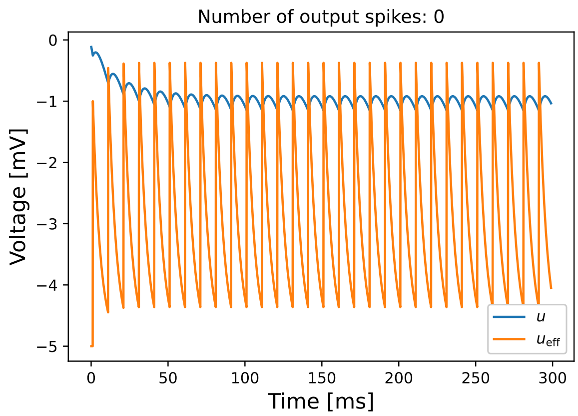


Figure 9: Weight 50, bias -3.0 mV, rate 80 Hz

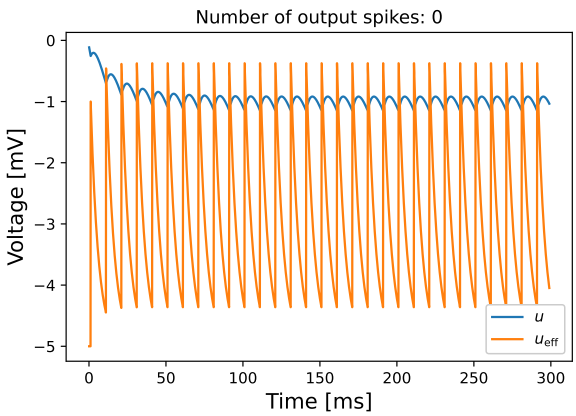


Figure 10: Weight 50, bias -3.0 mV, rate 100 Hz

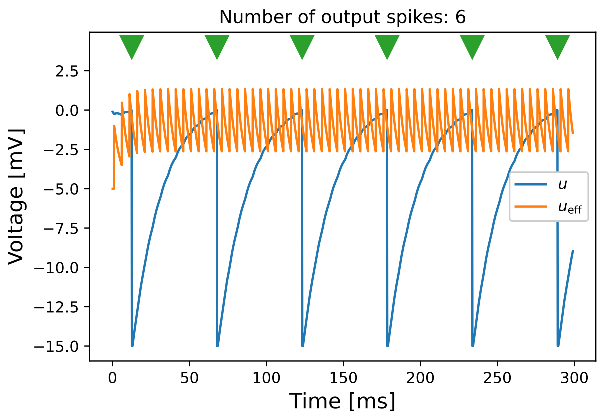


Figure 11: Weight 50, bias -3.0 mV, rate 200 Hz

As the name suggests, the weight is a quantity that gives "importance" to the input. The higher the weight, the more "importance" is given to the input.
If the weight is 0, the input is indeed ignored and the output stays at the bias value (remember to move u_{eff} up of 2 mV). The higher the weight, the higher the peak of the spikes, the stronger the "push" to the output, the easier to reach the spiking threshold.

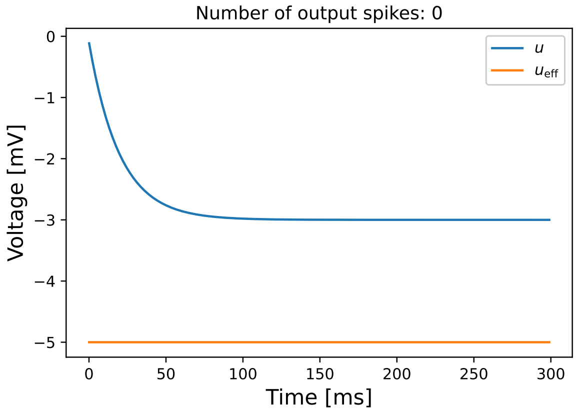


Figure 12: Weight 0, bias -3.0 mV, rate 100 Hz

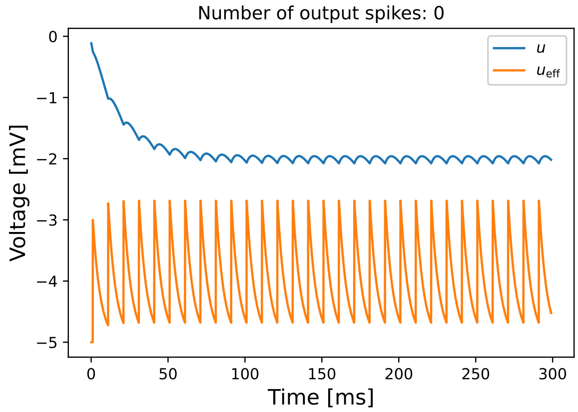


Figure 13: Weight 10, bias -3.0 mV, rate 100 Hz

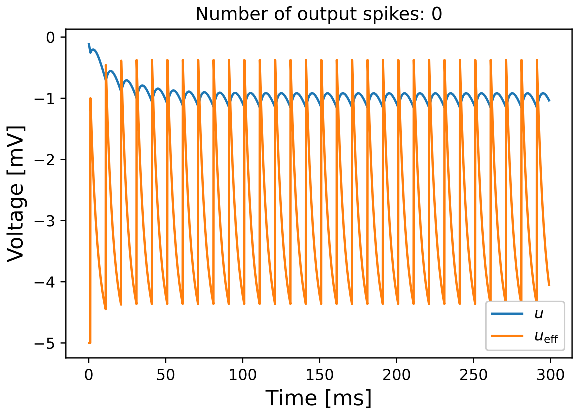


Figure 14: Weight 20, bias -3.0 mV, rate 100 Hz

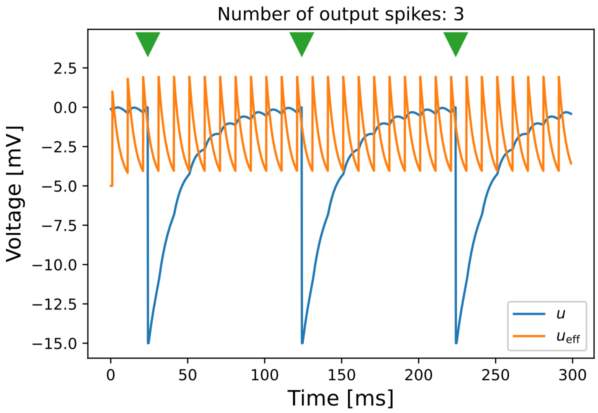


Figure 15: Weight 30, bias -3.0 mV, rate 100 Hz

The output is normally driven toward the bias value, when no input is provided. When an input is provided, the output voltage tries to reach u_{eff}. This means that the input decays untill u_{eff} is met, i.e. untill the blue and orange plots meet each other (the orange curve in the plots should be shifted up of 2 mV). Equilibrium is reached when there is a sort of resonance between the input frequency and the typical decay time of the output.

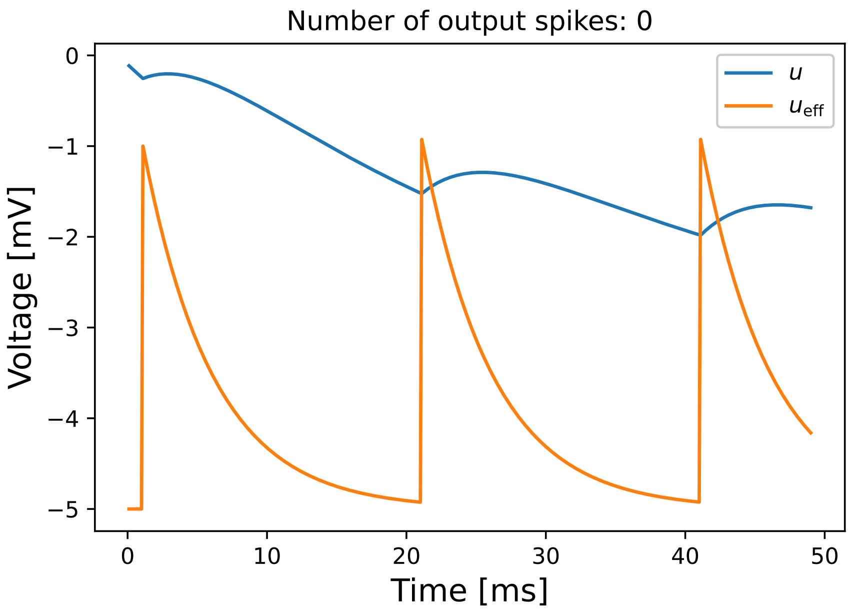


Figure 16: Weight 20, bias -3.0 mV, rate 50

c)

Overall the behaviour is the same as before, since the average values are the same. Thus, since now the peaks are not anymore equally spaced, two consecutives spikes may be so close each other that, combined to a "fortunate" previous state of the system, may lead the latter to spike in a configuration which did not lead to a spike in the regular case.

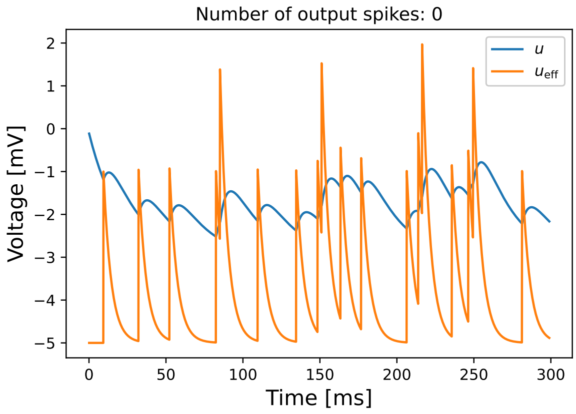


Figure 17: Weight 50, bias -3.0 mV, rate 50 Hz

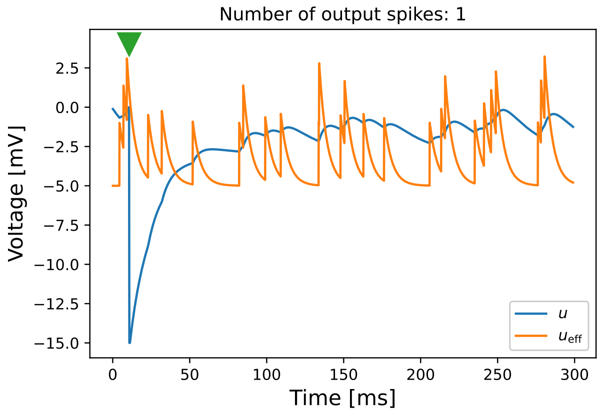


Figure 18: Weight 50, bias -3.0 mV, rate 80 Hz

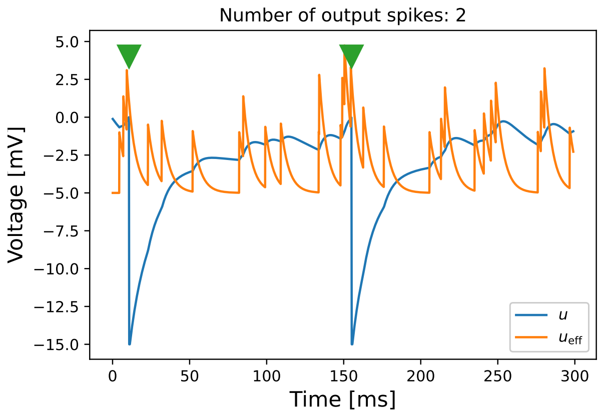


Figure 19: Weight 50, bias -3.0 mV, rate 100 Hz

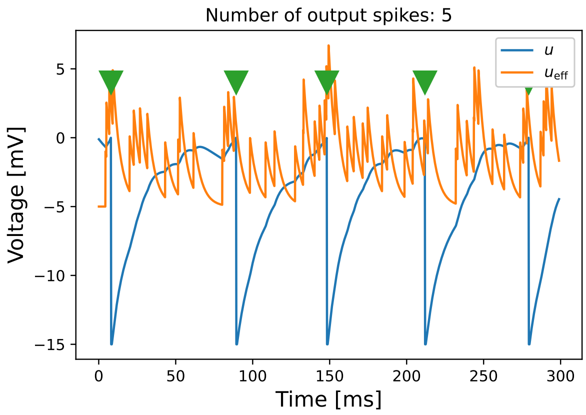


Figure 20: Weight 50, bias -3.0 mV, rate 200 Hz

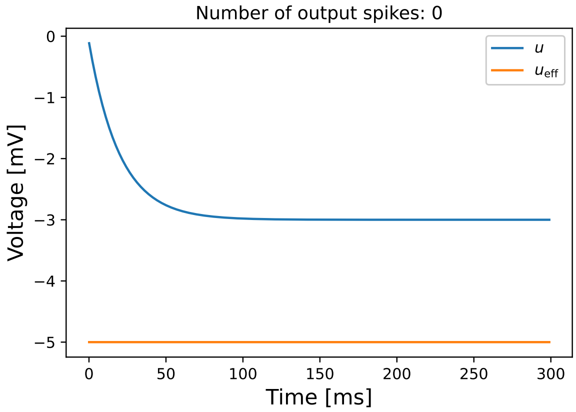


Figure 21: Weight 0, bias -3.0 mV, rate 100 Hz

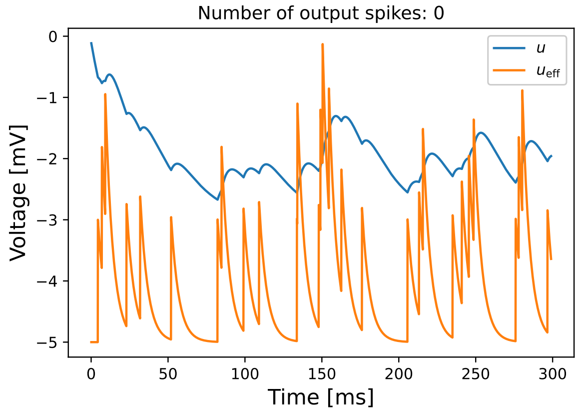


Figure 22: Weight 10, bias -3.0 mV, rate 100 Hz

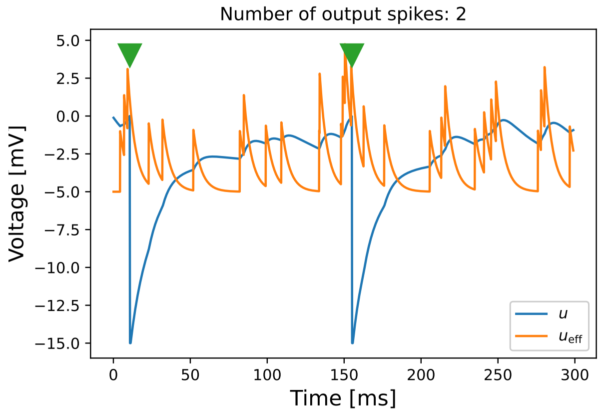


Figure 23: Weight 20, bias -3.0 mV, rate 100 Hz

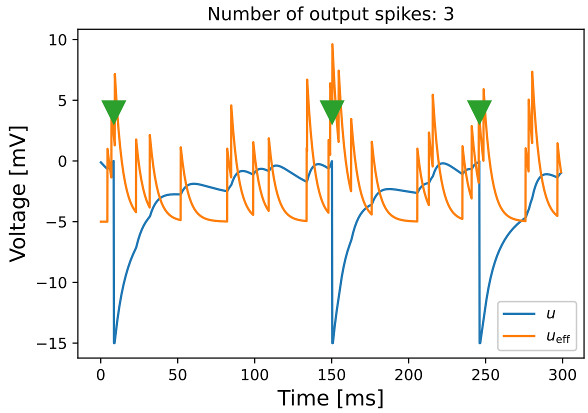


Figure 24: Weight 30, bias -3.0 mV, rate 100 Hz

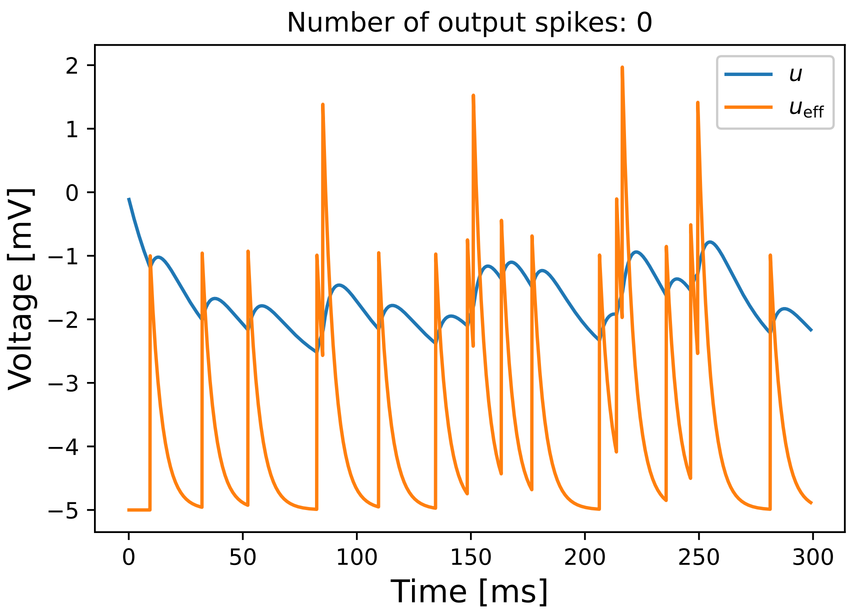


Figure 25: Weight 20,bias -3.0 mV, rate 50

Übung 5

Montag, 29. November 2021 11:51

Ex. 2

1) $\phi'(x) = ?$

with $\phi(x) = x \cdot \Theta(x)$ and $\Theta(x) =$ Heaviside step func.

$$\phi'(x) = 1 \cdot \Theta(x) + x \cdot \Theta'(x)$$

The derivative of the Heaviside function is the dirac delta function $\delta(x) = \begin{cases} +\infty, & x = 0 \\ 0, & x \neq 0 \end{cases}$

Therefore,

$$\phi'(x) = \Theta(x) + x \cdot \delta(x)$$

2) $\frac{dV(I)}{dI} = ?$

with $v(I) = \left(J_{ref} + J_m \cdot \log\left(\frac{E_{reset} - u_{eff}}{\Theta - u_{eff}} \right) \right)^{-1}$

and $u_{eff} = E_L + \frac{I}{gL}$

we can therefore rewrite $v(I)$ as:

$$v(I) = \Big(J_{ref} + \underbrace{J_m \cdot \left[\log\left(E_{reset} - E_L - \frac{I}{gL}\right) - \log\left(\Theta - E_L - \frac{I}{gL}\right) \right]}_{\text{using } a(I):} \Big)^{-1}$$

using $a(I)$:

$$\frac{dv(I)}{d(I)} = -1 \cdot [a(I)]^{-2} \cdot a'(I)$$

with:

$$a'(I) = \tau_m \left[\frac{-\frac{1}{gL}}{E_{reset} - E_L - \frac{I}{gL}} - \frac{-\frac{1}{gL}}{\Theta - E_L - \frac{I}{gL}} \right]$$

$$= \frac{\tau_m}{gL} \left[\frac{1}{\Theta - E_L - \frac{I}{gL}} - \frac{1}{E_{reset} - E_L - \frac{I}{gL}} \right]$$

Full equation:

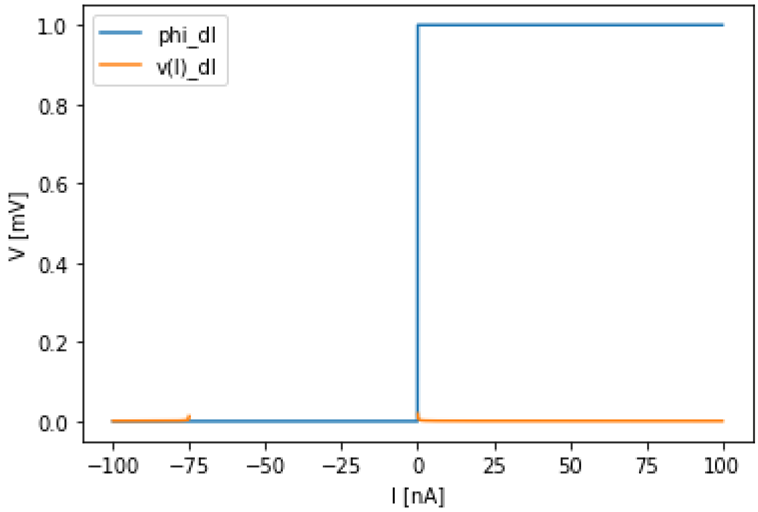
$$\frac{dv(t)}{d(I)} = \frac{-\frac{\tau_m}{gL}\left(\frac{1}{\Theta - E_L - \frac{I}{gL}} - \frac{1}{E_{reset} - E_L - \frac{I}{gL}} \right)}{\left[\tau_{ref} + \tau_m \cdot \log\left(\frac{E_{reset} - E_L - \frac{I}{gL}}{\Theta - E_L - \frac{I}{gL}} \right) \right]^2}$$

```
import matplotlib.pyplot as plt
import numpy as np


def phi_dx(x):

    # heaviside part
    if x > 0:
        h = 1.
    else:
        h = 0.

    # dirac delta part
    if x == 0:
        d = np.inf
    else:
        d = 0.

    return h + x * d


def dv_dI(I, nparams):

    # extract neuron params
    tau_m = nparams.get('tau_m')
    tau_ref = nparams.get('t_ref')
    V_th = nparams.get('V_th')
    V_reset = nparams.get('V_reset')
    E_L = nparams.get('E_L')
    g_L = nparams.get('C_m') / tau_m

    # calculate
    u1 = V_reset - E_L - I/g_L
    u2 = V_th - E_L - I/g_L
    numerator = - tau_m * (1/u2 - 1/u1) / g_L
    denominator = (tau_ref + tau_m * np.log(u1/u2) )**2

    return numerator/denominator


# define params
nparams={'C_m': 100., 'V_th':0., 'V_reset':-15., 'tau_m': 20.,
         'tau_syn_ex': 5., 'tau_syn_in': 5., 't_ref': 0.5, 'E_L': 0.,
         'V_m': -0.1}
I = np.linspace(-100., 100., 2000)
V_1 = np.zeros_like(I)
V_2 = np.zeros_like(I)


# get values
for i in range(I.size):
    V_1[i] = phi_dx(I[i])
    V_2[i] = dv_dI(I[i], nparams)
```

```
/usr/local/lib/python3.7/dist-packages/ipykernel_launcher.py:15: RuntimeWarning: invalid value
  from ipykernel import kernelapp as app
```

```
# plot
plt.plot(I, V_1, label='phi_dI')
plt.plot(I, V_2, label='v(I)_dI')
plt.xlabel('I [nA]')
plt.ylabel('V [mV]')
plt.legend()
```

```
<matplotlib.legend.Legend at 0x7f176e7fdad0>
```



Using the neuron parameters from the template both functions differ in their respone for any I above -75 nA. v(I)_dI seems to be not define between -75 and 0, the function values goes to inifinte for those two points. It is stable otherwise. In contrast to that, phi_dI is infinite at I = 0, but yields a stable 0 for any negative I or 1 for every positive I.

0 s Abgeschlossen um 13:38

Exercise 3

November 30, 2021

Exercise 5.3

a)

We Begin by writing down the XOR truth table:

$x1$	$x2$	out
0	0	0
1	0	1
0	1	1
1	1	0

The XOR gate takes two binary inputs and returns a binary output. That means that the input is a vector of two elements. We can also see that the cases are limited to 4. Thus the training set is limited to a collection of 4 bisimensional vectors.

b)

If we don't want to change E_l, we can just add a unitary bias to the input vector, thus incrementing its shape to $n = 3$.

c)

[19]:
```
import numpy as np
import matplotlib.pyplot as plt
from pprint import pprint
import nest
nest.set_verbosity("M_ERROR")
%matplotlib inline
```

Define neuron parameters and setup the dataset

[20]:
```
nparams = {'V_th': 0., 'V_reset': -1., 'tau_m': 40.,
           'tau_syn_ex': 5., 'tau_syn_in': 5.,
           't_ref': 0.5, 'E_L': -1.}

# define XOR data
data_inp = np.zeros((4, 2))
```

```
data_inp[1::2, 0] = 1
data_inp[2:, 1] = 1
data_cls = ((data_inp > 0.5).sum(axis=1) % 2) * 1
data_inp = np.hstack([data_inp, np.ones((len(data_inp), 1))])
print(data_inp)
print(data_cls)
```

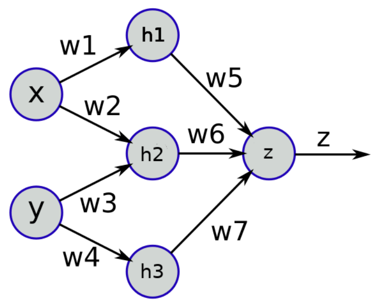
```
[[0. 0. 1.]
 [1. 0. 1.]
 [0. 1. 1.]
 [1. 1. 1.]]
[0 1 1 0]
```

[21]:

```
def run_feed_forward_network(inp, weights, maxrate=1000.,
                             duration=1000., nparams=nparams):
    """Execute a single feed forward network.

    Input:
        inp      array           either a single input vector or an
                                 array of input vectors
        weights  list of arrays  list of the weight matrices between
                                 the different layers
        maxrate  float           rate of the spike sources that corresponds
                                 to input =1
        duration float           duration of the simulation per image
        nparams  dict            neuron parameters

    Output:
        spike_rates  list        list of the the spike rate arrays per layer
    """
    # Reset NEST
    nest.ResetKernel()

    # If only one input example is given: Put it into a
    # (1, ninput) array so that the iteration gives only
    # 1 run
    if len(inp.shape) == 1:
        inp = inp.reshape(1, -1)

    # create spike sources
    num_inp = weights[0].shape[1]
    spikegenerators = nest.Create('poisson_generator', num_inp)

    # generate all neuron layers by iterating over the list of weights
    hiddens, spikedetectors = [], []
    for i in range(len(weights)):
        # create neurons for this layer and record their spikes
```

```
        neurons = weights[i].shape[0]
        hiddens.append(
            nest.Create('iaf_psc_exp', neurons, params=nparams)
        )
        spikedetectors.append(
            nest.Create('spike_recorder', neurons)
        )
        nest.Connect(hiddens[-1], spikedetectors[-1], 'one_to_one')

        # in the first layer get input from the spikesources, else from
        # the previous layer, which is the second to last in the hiddens
        # array
        if i == 0:
            presyn = spikegenerators
        else:
            presyn = hiddens[-2]
        # connect sources with appropriate weights
        nest.Connect(presyn, hiddens[-1], syn_spec={'weight': weights[i]})

    spike_rates = []
    for inpimg in inp:
        # set up simulation and equilibrate system
        nest.SetStatus(spikegenerators, {'rate': inpimg * maxrate})
        nest.Simulate(100.)
        # Reset spike counters
        for i in range(len(weights)):
            spikedetectors[i].n_events = 0

        # Do the actual simulation
        nest.Simulate(duration)

        # read out spikes
        spike_rates.append([])
        for i in range(len(weights)):
            spike_rates[-1].append(
                np.array(nest.GetStatus(spikedetectors[i], "n_events")) * 1000. /
↪ duration / maxrate
            )

    return spike_rates
```

We consider a three layer network, similar to the one shown below (the interface will give you all-too-all connectivity). Note that all activities are rescaled to the inputrate which you can set via the `maxrate` parameter. Compare this to the membrane and synaptic time constants choosen above.



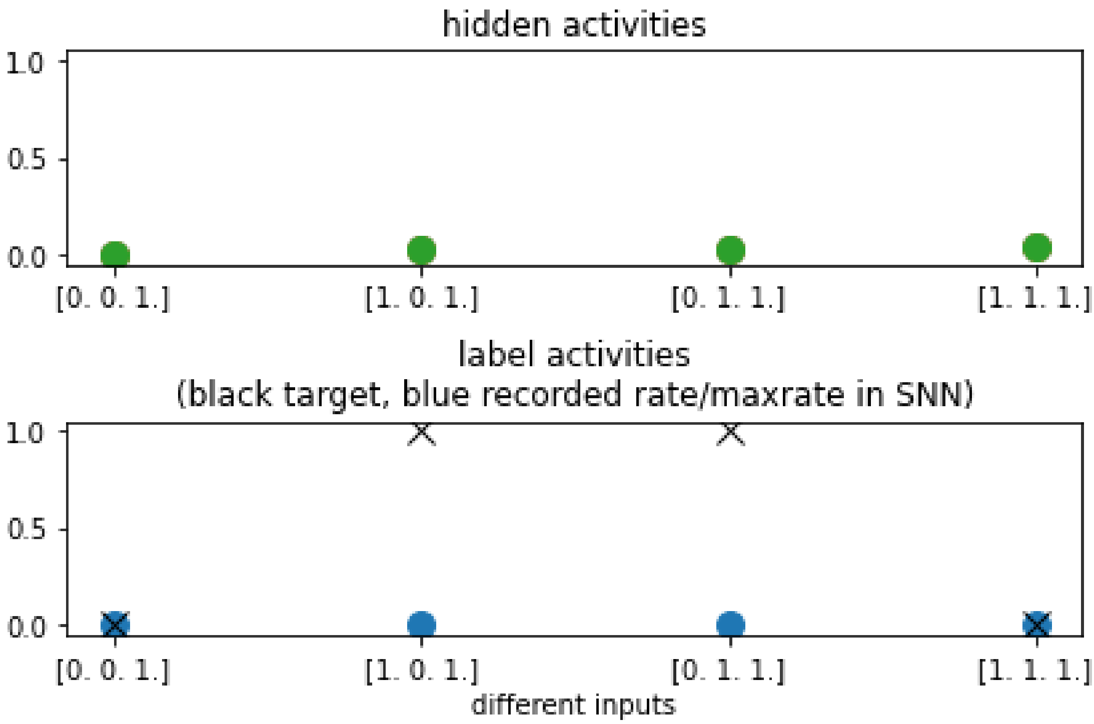
[22]:
```
def run_xor(W_IH = np.array([[1., 0., 0.], [0., 1., 0.], [1., 1., -1.],]) * 100.,
            W_HL = np.array([[1, 1, -3],]) * 100.,
            maxrate = 1000.,
            duration = 1000.,
            ):
    """Execute the XOR network

    Inputs:
        W_IH     array   weight matrix from input to hidden layer
        W_HL     array   weight matrix from hidden to output layer
        maxrate  float   rate of the spike sources that corresponds
                         to input =1
    """
    fig, axes = plt.subplots(2, 1)

    # run the network for all inputs, defined above
    # (for later tasks you may want to make this a parameter)
    result = run_feed_forward_network(
        data_inp,
        [
            W_IH,
            W_HL,
        ],
        maxrate=maxrate,
        nparams=nparams
    )

    # plot the activities of both hidden (axes[0]) and output (axes[1]) layers
    # for the different points of the dataset (x-axis)
    for i in range(len(data_inp)):
        for j in range(len(result[i][0])):
            axes[0].plot([i], result[i][0][j], c=f"C{j}", marker='o',␣
↪markersize=10)

        for j in range(len(result[i][1])):
            axes[1].plot([i], result[i][1][j], c=f"C{j}", marker='o',␣
↪markersize=10)
```

```
        axes[1].plot(i, data_cls[i], c="black", marker='x', markersize=10)

    axes[0].set_title("hidden activities")
    axes[0].set_xticks(range(len(data_inp)))
    axes[0].set_xticklabels([str(d) for d in data_inp])
    axes[0].set_ylim(-0.05, 1.05)

    axes[1].set_title("label activities\n(black target, blue recorded rate/
↪maxrate in SNN)")
    axes[1].set_xlabel("different inputs")
    axes[1].set_xticks(range(len(data_inp)))
    axes[1].set_xticklabels([str(d) for d in data_inp])
    axes[1].set_ylim(-0.05, 1.05)

    fig.tight_layout()
    print("result")
    pprint([res[0] for res in result])
    pprint([res[1] for res in result])
```

We now play a bit with the weigts scale to see how it affects the activities of the network:

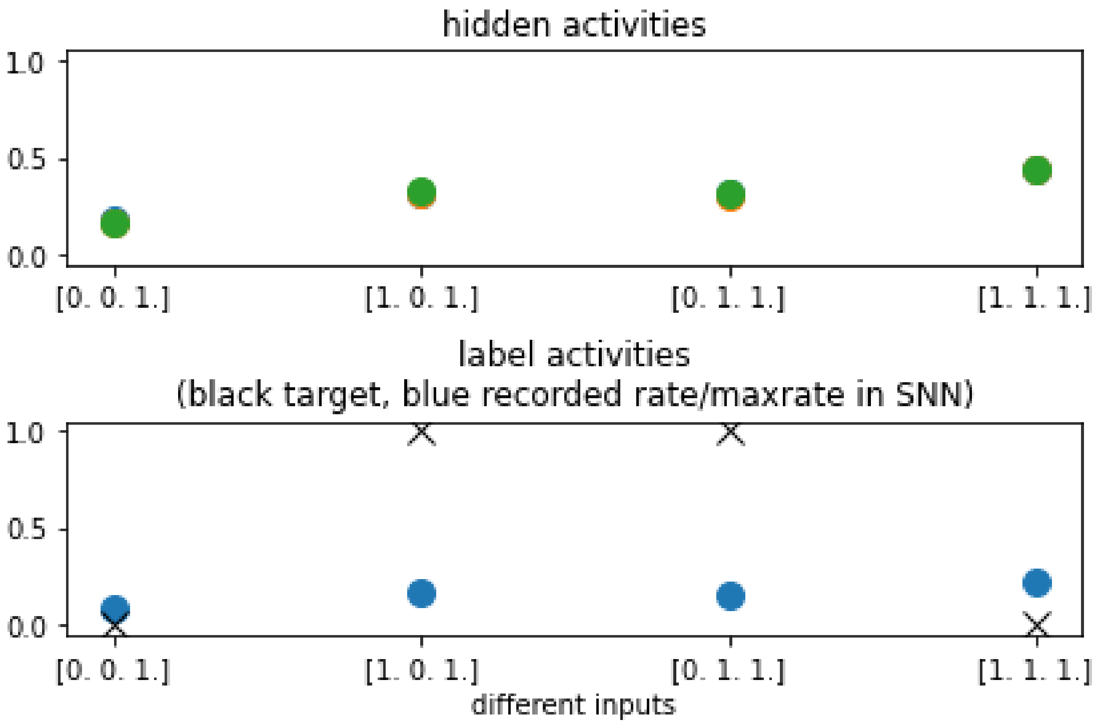
[23]:
```
weight_scale = 1.
W_IH = weight_scale * np.array(
    [[1, 1, 1],
     [1, 1, 1],
     [1, 1, 1],
    ])
W_HL = weight_scale * np.array([[1, 1, 1],])
run_xor(W_IH, W_HL, maxrate=1000., duration=1000.)
```

```
result
[array([0.001, 0.   , 0.   ]),
 array([0.025, 0.024, 0.026]),
 array([0.025, 0.024, 0.024]),
 array([0.045, 0.045, 0.046])]
[array([0.]), array([0.]), array([0.]), array([0.])]
```



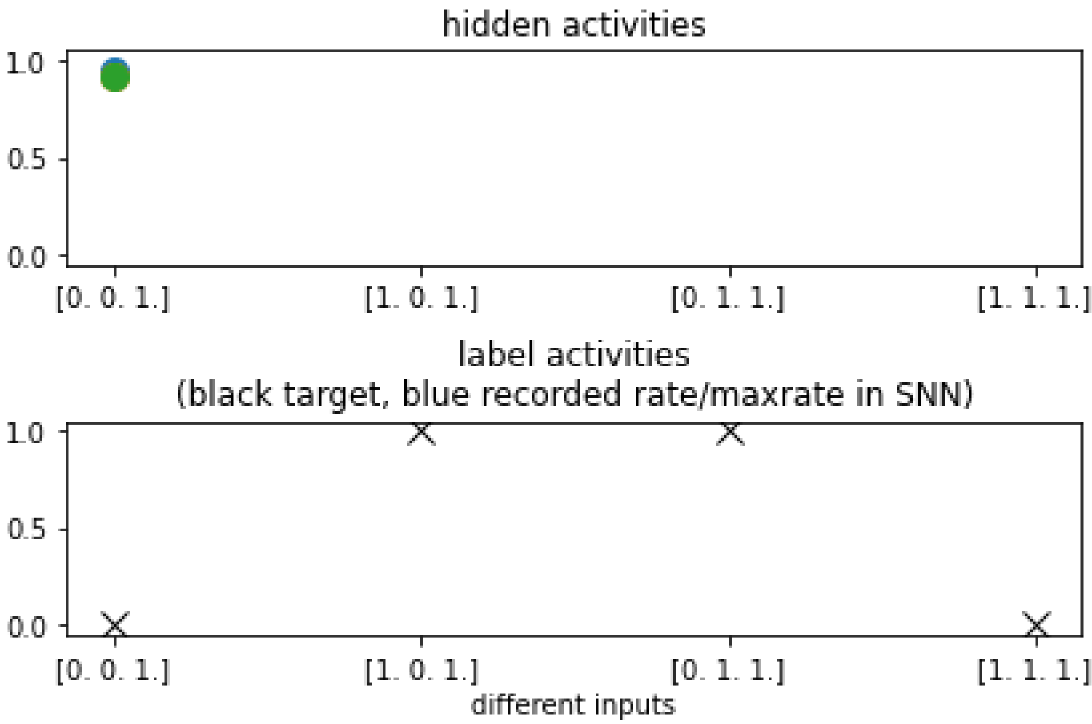
[24]:
```
weight_scale = 10.
W_IH = weight_scale * np.array(
    [[1, 1, 1],
     [1, 1, 1],
     [1, 1, 1],
    ])
W_HL = weight_scale * np.array([[1, 1, 1],])
run_xor(W_IH, W_HL, maxrate=1000., duration=1000.)
```

```
result
[array([0.174, 0.166, 0.166]),
 array([0.316, 0.31 , 0.326]),
 array([0.315, 0.307, 0.314]),
 array([0.443, 0.443, 0.443])]
[array([0.08]), array([0.162]), array([0.159]), array([0.222])]
```



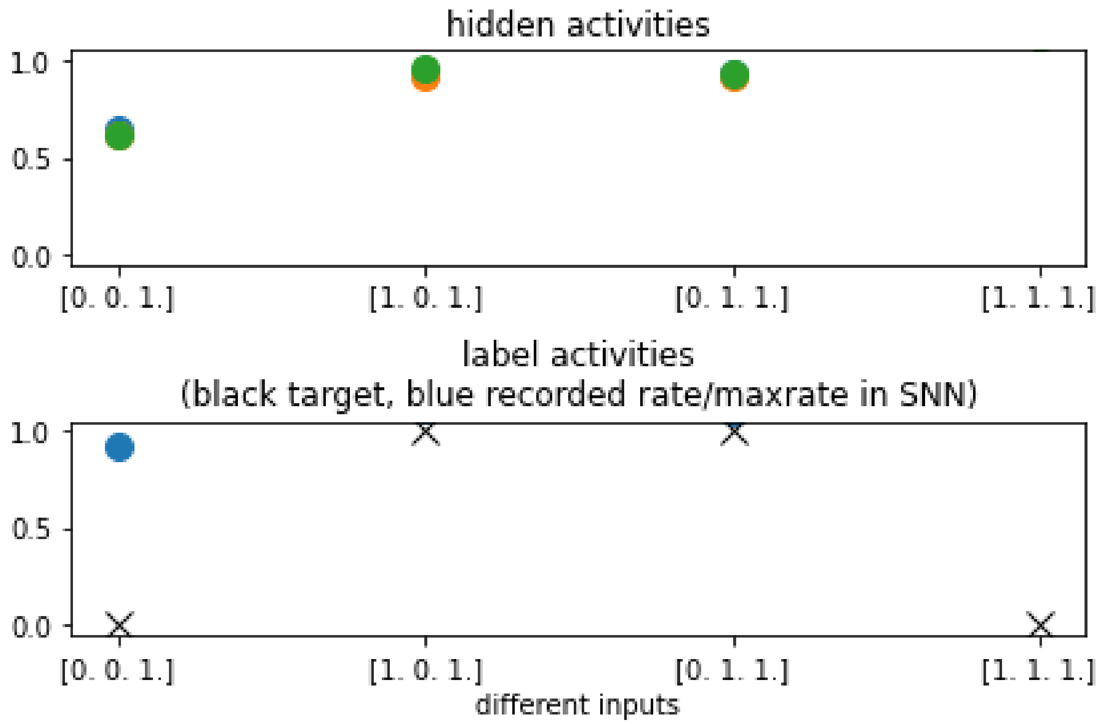
[25]:
```
weight_scale = 100.
W_IH = weight_scale * np.array(
    [[1, 1, 1],
     [1, 1, 1],
     [1, 1, 1],
    ])
W_HL = weight_scale * np.array([[1, 1, 1],])
run_xor(W_IH, W_HL, maxrate=1000., duration=1000.)
```

```
result
[array([0.944, 0.915, 0.918]),
 array([1.23 , 1.222, 1.25 ]),
 array([1.229, 1.221, 1.23 ]),
 array([1.394, 1.397, 1.391])]
[array([1.407]), array([1.428]), array([1.429]), array([1.428])]
```



[26]:
```
weight_scale = 50.
W_IH = weight_scale * np.array(
    [[1, 1, 1],
     [1, 1, 1],
     [1, 1, 1],
    ])
W_HL = weight_scale * np.array([[1, 1, 1],])
run_xor(W_IH, W_HL, maxrate=1000., duration=1000.)
```

```
result
[array([0.642, 0.618, 0.619]),
 array([0.934, 0.924, 0.953]),
 array([0.933, 0.92 , 0.932]),
 array([1.126, 1.124, 1.124])]
[array([0.918]), array([1.107]), array([1.104]), array([1.185])]
```



We can see that increasing the weights scale we also increase the magnitude both of the hidden layer and of the final output. If we set it to a too small value, all the outputs will be null, while if we set it to a too large value, the outputs will be all over the targets. Thus we have to choose a reasonable intermediate value, that has to be adjusted with respect to the maxrate and to the different weights sets.

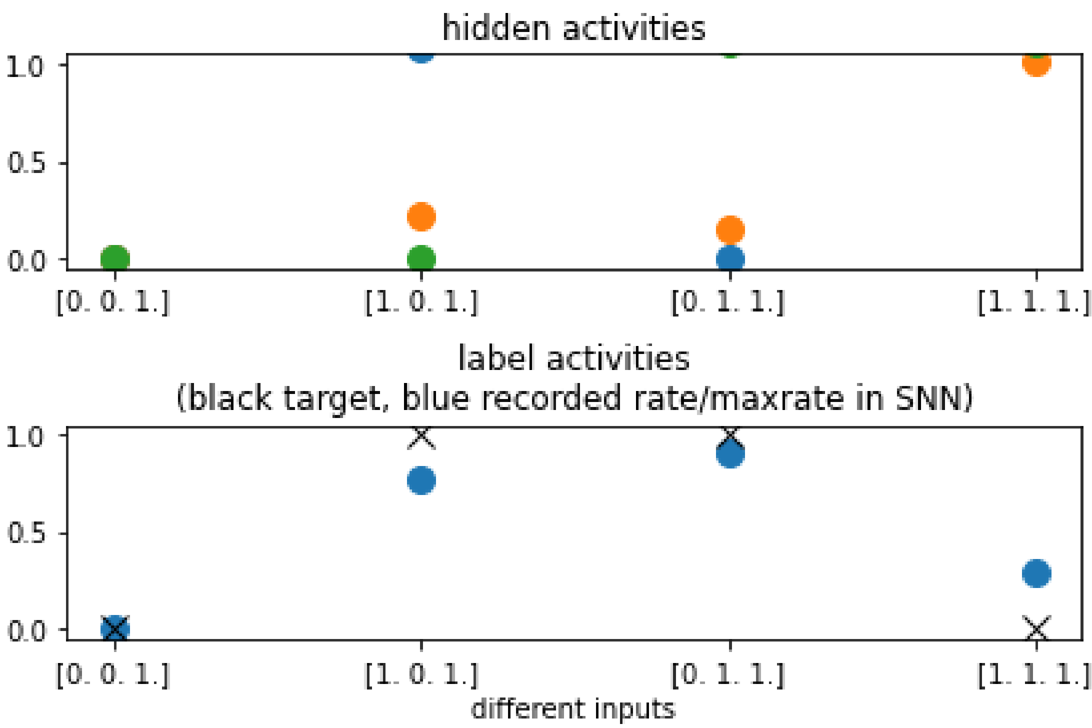
d)

We will choose the simplest binarisation scheme for the output: 1 if the output y is over a threshold Θ and 0 if $y \leq \Theta$. In our case, we will set $\Theta = 0.5$. We then try using the weights from the previous sheet, and, adjusting a bit the scale, we obtain a corret classification:

[27]:
```
weight_scale = 150
W_IH = weight_scale * np.array(
    [[1, 0, 0],
     [1, 1, -1],
     [0, 1, 0],
    ])
W_HL = weight_scale * np.array([[1, -2, 1],])
run_xor(W_IH, W_HL, maxrate=1000., duration=1000.)
```

```
result
[array([0., 0., 0.]),
 array([1.078, 0.218, 0.   ]),
```

```
 array([0.   , 0.153, 1.116]),
 array([1.107, 1.012, 1.109])]
[array([0.]), array([0.766]), array([0.911]), array([0.29])]
```



e)

i) AND gate:

[28]:
```
nparams = {'V_th': 0., 'V_reset': -1., 'tau_m': 40.,
           'tau_syn_ex': 5., 'tau_syn_in': 5.,
           't_ref': 0.5, 'E_L': -1.}

# define AND data
data_inp = np.zeros((4, 2))
data_inp[1::2, 0] = 1
data_inp[2:, 1] = 1
data_cls = [0,0,0,1]
data_inp = np.hstack([data_inp, np.ones((len(data_inp), 1))])
print(data_inp)
print(data_cls)
```

```
[[0. 0. 1.]
 [1. 0. 1.]
 [0. 1. 1.]
```

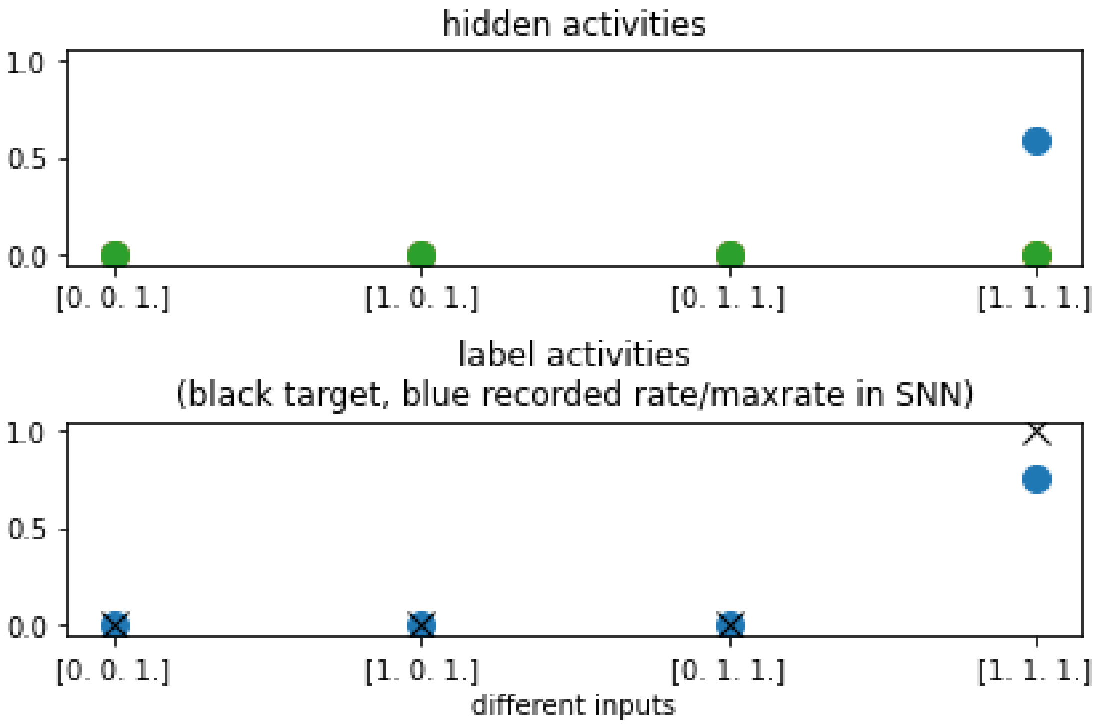
```
 [1. 1. 1.]]
[0, 0, 0, 1]
```

[29]:
```python
def run_and(W_IH = np.array([[1., 0., 0.], [0., 1., 0.], [1., 1., -1.],]) * 100.,
            W_HL = np.array([[1, 1, -3],]) * 100.,
            maxrate = 1000.,
            duration = 1000.,
            ):
    """Execute the AND network

    Inputs:
        W_IH     array   weight matrix from input to hidden layer
        W_HL     array   weight matrix from hidden to output layer
        maxrate  float   rate of the spike sources that corresponds
                         to input =1
    """
    fig, axes = plt.subplots(2, 1)

    # run the network for all inputs, defined above
    # (for later tasks you may want to make this a parameter)
    result = run_feed_forward_network(
        data_inp,
        [
            W_IH,
            W_HL,
        ],
        maxrate=maxrate,
        nparams=nparams
    )

    # plot the activities of both hidden (axes[0]) and output (axes[1]) layers
    # for the different points of the dataset (x-axis)
    for i in range(len(data_inp)):
        for j in range(len(result[i][0])):
            axes[0].plot([i], result[i][0][j], c=f"C{j}", marker='o',␣
↪markersize=10)

        for j in range(len(result[i][1])):
            axes[1].plot([i], result[i][1][j], c=f"C{j}", marker='o',␣
↪markersize=10)
        axes[1].plot(i, data_cls[i], c="black", marker='x', markersize=10)

    axes[0].set_title("hidden activities")
    axes[0].set_xticks(range(len(data_inp)))
    axes[0].set_xticklabels([str(d) for d in data_inp])
    axes[0].set_ylim(-0.05, 1.05)
```

```
    axes[1].set_title("label activities\n(black target, blue recorded rate/
↪maxrate in SNN)")
    axes[1].set_xlabel("different inputs")
    axes[1].set_xticks(range(len(data_inp)))
    axes[1].set_xticklabels([str(d) for d in data_inp])
    axes[1].set_ylim(-0.05, 1.05)

    fig.tight_layout()
    print("result")
    pprint([res[0] for res in result])
    pprint([res[1] for res in result])
```

[30]:
```
weight_scale = 150
W_IH = weight_scale * np.array(
    [[1, 1, -1.5],
     [0, 0, -0],
     [0, 0, 0],
    ])
W_HL = weight_scale * np.array([[1, -2, 1],])
run_and(W_IH, W_HL, maxrate=1000., duration=1000.)
```

```
result
[array([0., 0., 0.]),
 array([0., 0., 0.]),
 array([0., 0., 0.]),
 array([0.592, 0.   , 0.   ])]
[array([0.]), array([0.]), array([0.]), array([0.757])]
```



ii) OR gate:

[31]:
```
nparams = {'V_th': 0., 'V_reset': -1., 'tau_m': 40.,
           'tau_syn_ex': 5., 'tau_syn_in': 5.,
           't_ref': 0.5, 'E_L': -1.}

# define OR data
data_inp = np.zeros((4, 2))
data_inp[1::2, 0] = 1
data_inp[2:, 1] = 1
data_cls = [0,1,1,1]
data_inp = np.hstack([data_inp, np.ones((len(data_inp), 1))])
print(data_inp)
print(data_cls)
```

```
[[0. 0. 1.]
 [1. 0. 1.]
 [0. 1. 1.]
 [1. 1. 1.]]
[0, 1, 1, 1]
```

[32]:
```
def run_or(W_IH = np.array([[1., 0., 0.], [0., 1., 0.], [1., 1., -1.],]) * 100.,
            W_HL = np.array([[1, 1, -3],]) * 100.,
```

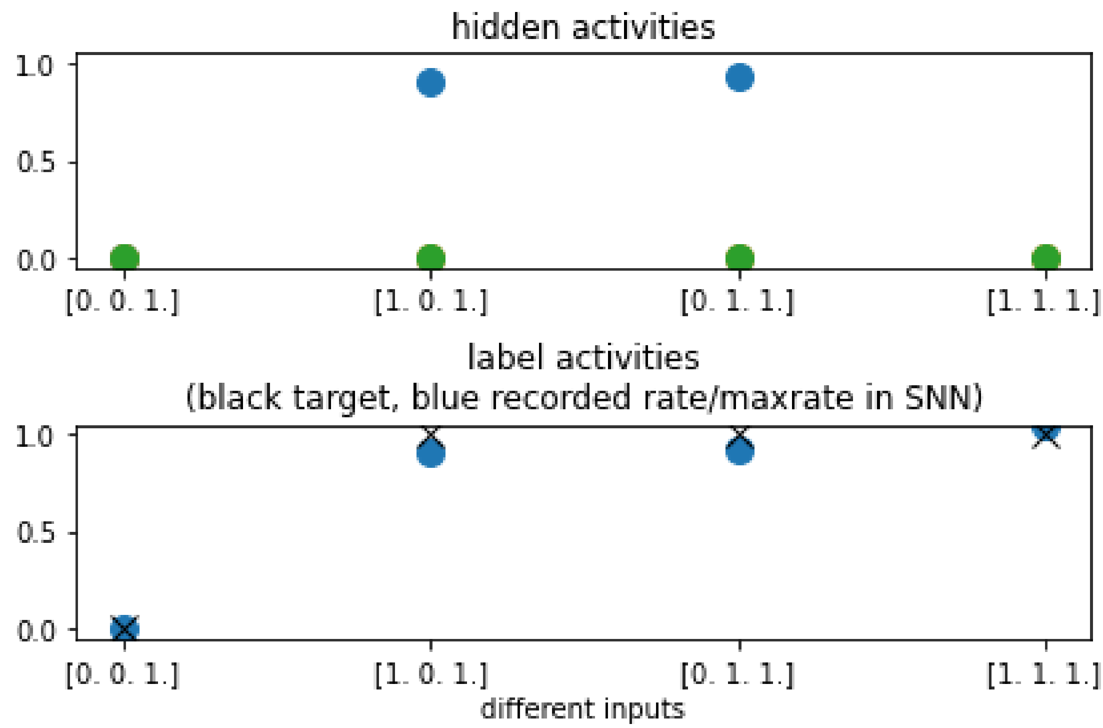
```
            maxrate = 1000.,
            duration = 1000.,
            ):
    """Execute the OR network

    Inputs:
        W_IH      array   weight matrix from input to hidden layer
        W_HL      array   weight matrix from hidden to output layer
        maxrate   float   rate of the spike sources that corresponds
                          to input =1
    """
    fig, axes = plt.subplots(2, 1)

    # run the network for all inputs, defined above
    # (for later tasks you may want to make this a parameter)
    result = run_feed_forward_network(
        data_inp,
        [
            W_IH,
            W_HL,
        ],
        maxrate=maxrate,
        nparams=nparams
    )

    # plot the activities of both hidden (axes[0]) and output (axes[1]) layers
    # for the different points of the dataset (x-axis)
    for i in range(len(data_inp)):
        for j in range(len(result[i][0])):
            axes[0].plot([i], result[i][0][j], c=f"C{j}", marker='o',␣
↪markersize=10)

        for j in range(len(result[i][1])):
            axes[1].plot([i], result[i][1][j], c=f"C{j}", marker='o',␣
↪markersize=10)
        axes[1].plot(i, data_cls[i], c="black", marker='x', markersize=10)

    axes[0].set_title("hidden activities")
    axes[0].set_xticks(range(len(data_inp)))
    axes[0].set_xticklabels([str(d) for d in data_inp])
    axes[0].set_ylim(-0.05, 1.05)

    axes[1].set_title("label activities\n(black target, blue recorded rate/
↪maxrate in SNN)")
    axes[1].set_xlabel("different inputs")
    axes[1].set_xticks(range(len(data_inp)))
    axes[1].set_xticklabels([str(d) for d in data_inp])
```

```python
    axes[1].set_ylim(-0.05, 1.05)

    fig.tight_layout()
    print("result")
    pprint([res[0] for res in result])
    pprint([res[1] for res in result])
```

[33]:

```python
weight_scale = 100
W_IH = weight_scale * np.array(
    [[1, 1, 0],
     [0, 0, 0],
     [0, 0, 0],
    ])
W_HL = weight_scale * np.array([[1, -2, 1],])
run_or(W_IH, W_HL, maxrate=1000., duration=1000.)
```

```
result
[array([0., 0., 0.]),
 array([0.901, 0.   , 0.   ]),
 array([0.938, 0.   , 0.   ]),
 array([1.236, 0.   , 0.   ])]
[array([0.]), array([0.9]), array([0.922]), array([1.05])]
```



iii) NOT gate:

Differently from the previous tasks, here we deal with a monodimensional input:

[34]:
```
nparams = {'V_th': 0., 'V_reset': -1., 'tau_m': 40.,
           'tau_syn_ex': 5., 'tau_syn_in': 5.,
           't_ref': 0.5, 'E_L': -1.}

# define XOR data
data_inp = np.zeros((2, 1))
data_inp = [[0], [1]]
data_cls = [1, 0]
data_inp = np.hstack([data_inp, np.ones((len(data_inp), 1))])
print(data_inp)
print(data_cls)
```

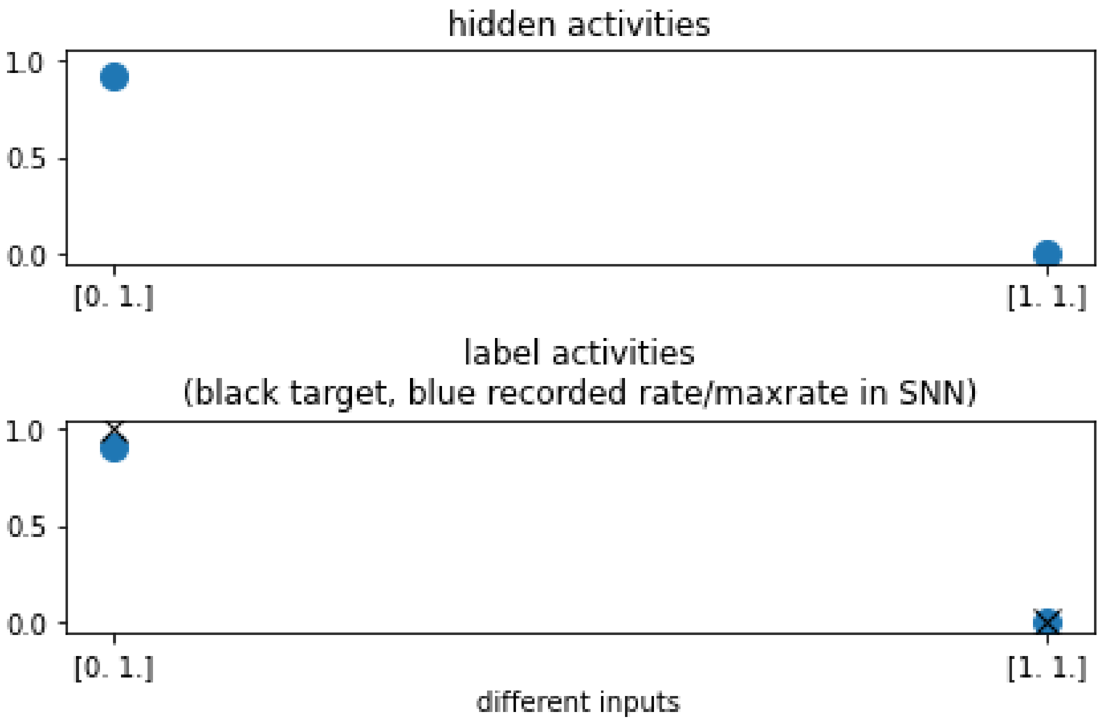
```
[[0. 1.]
 [1. 1.]]
[1, 0]
```

[35]:
```
def run_not(W_IH = np.array([[1., 0., 0.], [0., 1., 0.], [1., 1., -1.],]) * 100.,
            W_HL = np.array([[1, 1, -3],]) * 100.,
            maxrate = 1000.,
            duration = 1000.,
            ):
    """Execute the NOT network

    Inputs:
        W_IH      array   weight matrix from input to hidden layer
        W_HL      array   weight matrix from hidden to output layer
        maxrate   float   rate of the spike sources that corresponds
                          to input =1
    """
    fig, axes = plt.subplots(2, 1)

    # run the network for all inputs, defined above
    # (for later tasks you may want to make this a parameter)
    result = run_feed_forward_network(
        data_inp,
        [
            W_IH,
            W_HL,
        ],
        maxrate=maxrate,
        nparams=nparams
    )

    # plot the activities of both hidden (axes[0]) and output (axes[1]) layers
```

```
    # for the different points of the dataset (x-axis)
    for i in range(len(data_inp)):
        for j in range(len(result[i][0])):
            axes[0].plot([i], result[i][0][j], c=f"C{j}", marker='o',␣
↪markersize=10)

        for j in range(len(result[i][1])):
            axes[1].plot([i], result[i][1][j], c=f"C{j}", marker='o',␣
↪markersize=10)
        axes[1].plot(i, data_cls[i], c="black", marker='x', markersize=10)

    axes[0].set_title("hidden activities")
    axes[0].set_xticks(range(len(data_inp)))
    axes[0].set_xticklabels([str(d) for d in data_inp])
    axes[0].set_ylim(-0.05, 1.05)

    axes[1].set_title("label activities\n(black target, blue recorded rate/
↪maxrate in SNN)")
    axes[1].set_xlabel("different inputs")
    axes[1].set_xticks(range(len(data_inp)))
    axes[1].set_xticklabels([str(d) for d in data_inp])
    axes[1].set_ylim(-0.05, 1.05)

    fig.tight_layout()
    print("result")
    pprint([res[0] for res in result])
    pprint([res[1] for res in result])
```

[36]:
```
weight_scale = 100
W_IH = weight_scale * np.array(
    [[-1.5, 1]])
W_HL = weight_scale * np.array([[1],])
run_not(W_IH, W_HL, maxrate=1000., duration=1000.)
```

```
result
[array([0.922]), array([0.])]
[array([0.911]), array([0.])]
```



f)

We just have to pass X1 and X2 to our trained XOR gate, and X3 and X4 to the AND gate. Then we pass the two outputs to the OR gate. If every gate is trained separately, the training is stable.

[ ]:

```
```