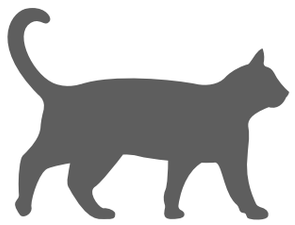
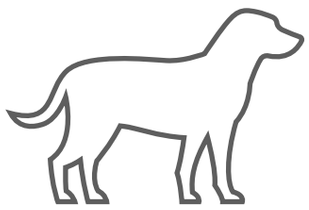
NEST::
memory capacity of SNNs

Swathi’s part

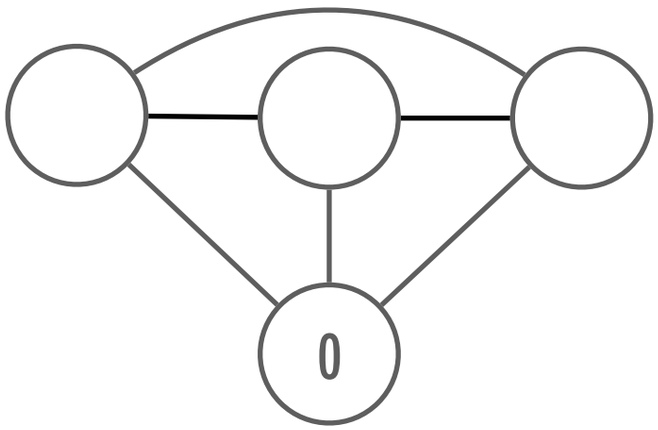
introduction background

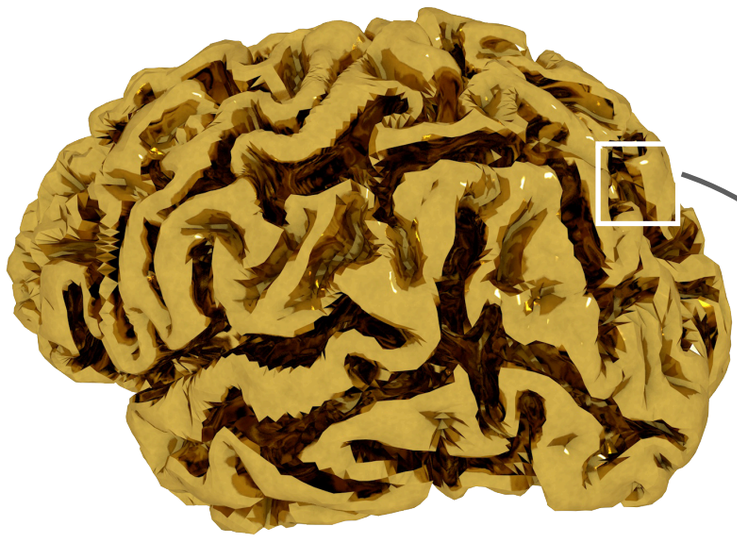


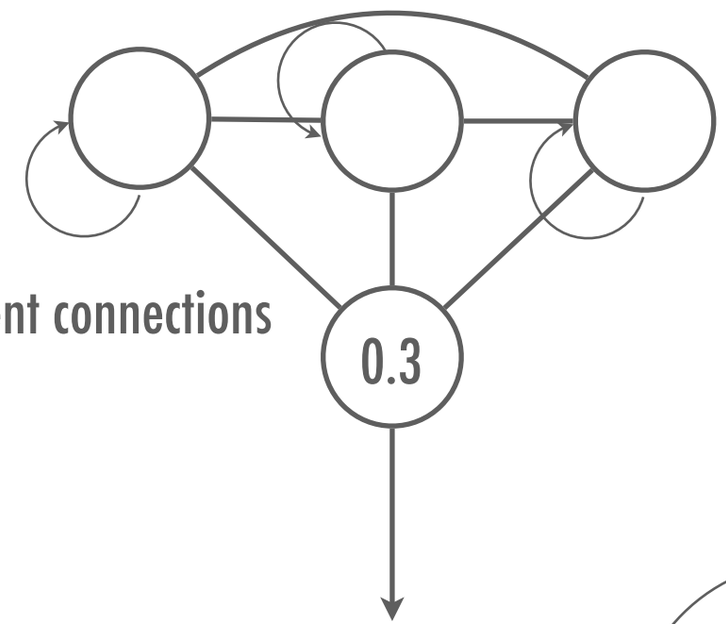
0



1





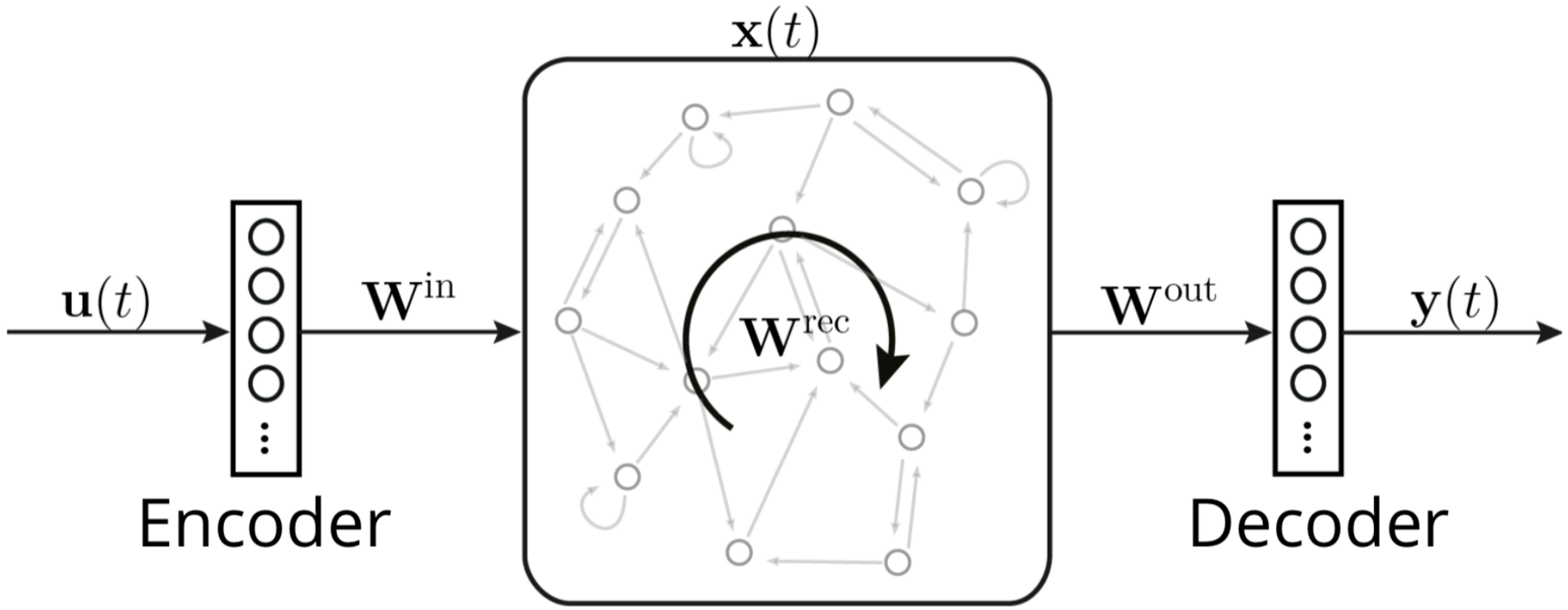


Why??

recurrent connections in the cortex

memory ?

Background

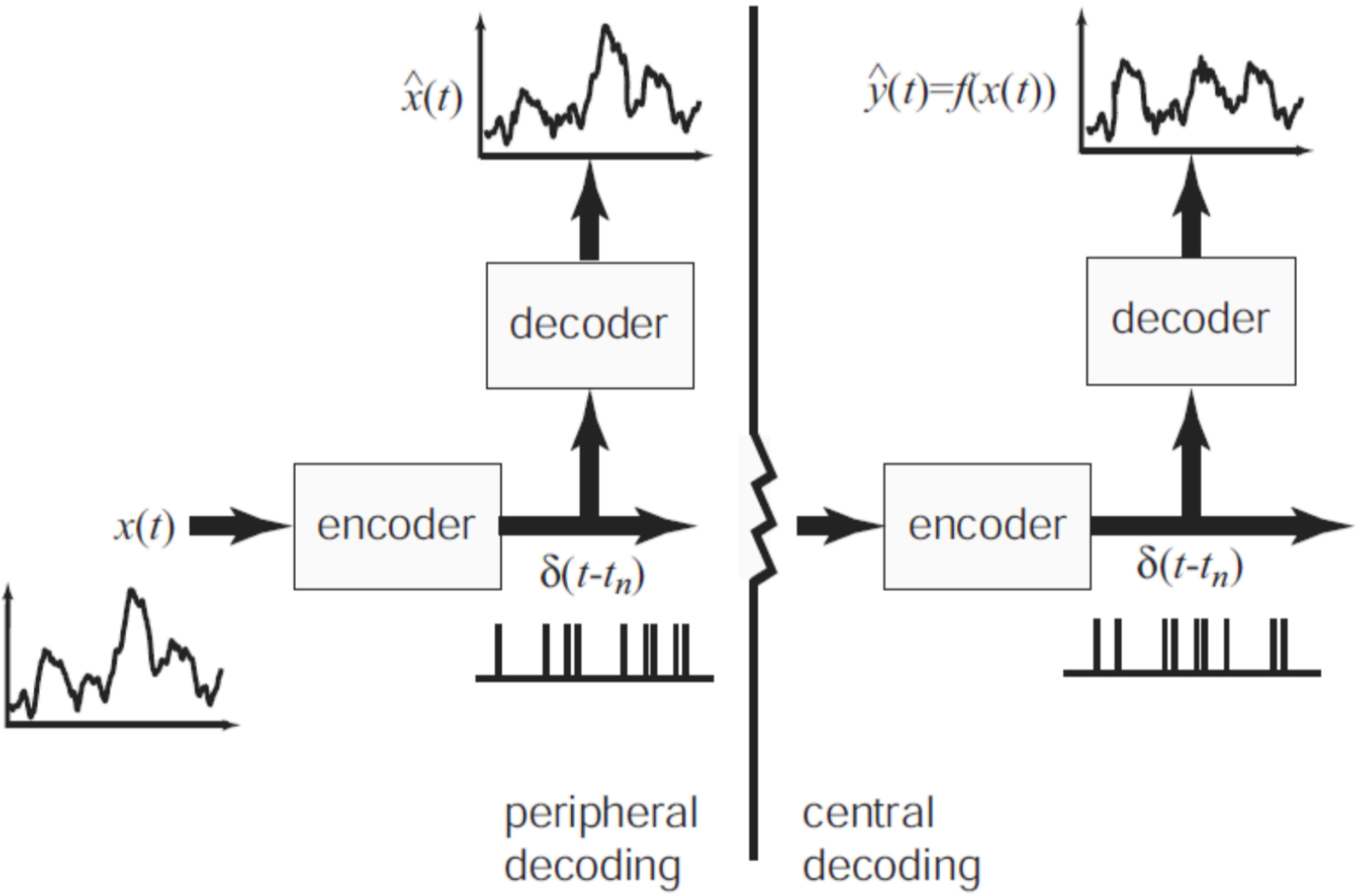


Circuit

$$\mathbf{u}(t) = \begin{bmatrix} u_1(t) \\ u_2(t) \\ \dots \\ u_{\mathrm{N}_u}(t) \end{bmatrix} \xrightarrow{\mathbf{W}^{\text{in}} \in \mathbb{R}_{\mathrm{N}_u \times \mathrm{N}_x}} \mathbf{x}(t) = \begin{bmatrix} x_1(t) \\ x_2(t) \\ \dots \\ x_{\mathrm{N}}(t) \end{bmatrix} \xrightarrow{\mathbf{W}^{\text{out}} \in \mathbb{R}_{\mathrm{N} \times \mathrm{N}_y}} \mathbf{y}(t) = \begin{bmatrix} y_1(t) \\ y_2(t) \\ \dots \\ y_{\mathrm{N}_y}(t) \end{bmatrix}$$

$$\mathbf{W}^{\text{rec}} \in \mathbb{R}_{\mathrm{N} \times \mathrm{N}}$$

Encoding



Encode an input signal into the activity of a layer of spiking neurons

Serve as input to the main processing circuit (RNN)

Eliasmith, C. and Anderson, C. (2003). *Neural Engineering: Computation, Representation and Dynamics in neurobiological systems.* MIT Press.

Decoding

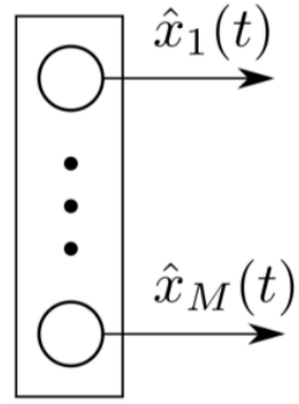
decoding → linear readout

Linear readout of population activity to reconstruct target signal

$$x(t), x(t-1), x(t-2), ...x(t-\tau)$$

decoders (synaptic weights) →

$$\boldsymbol{D} = \begin{pmatrix} \vec{d}_1 \\ \vdots \\ \vec{d}_N \end{pmatrix} = \boldsymbol{\Gamma}^{-1}\boldsymbol{\Upsilon}$$

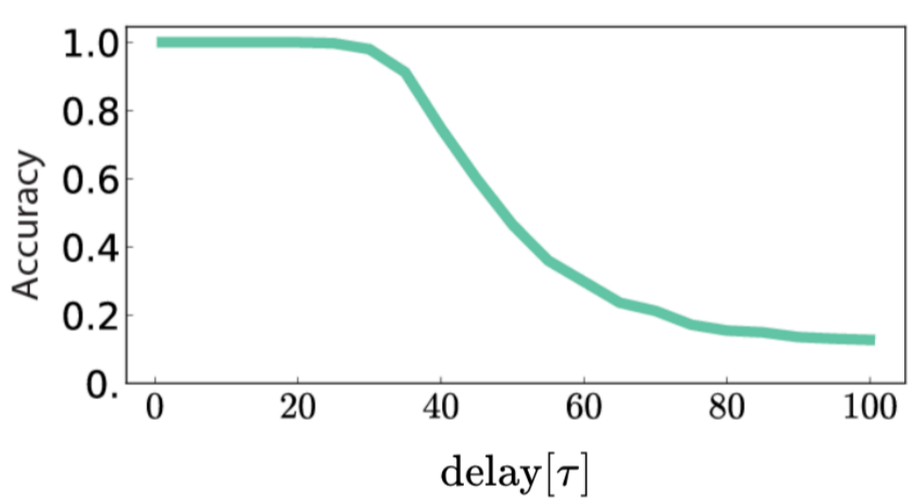


$$\hat{\vec{x}}(t) = \sum_{i=1}^{N} a_i(t)\vec{d}_i$$

with

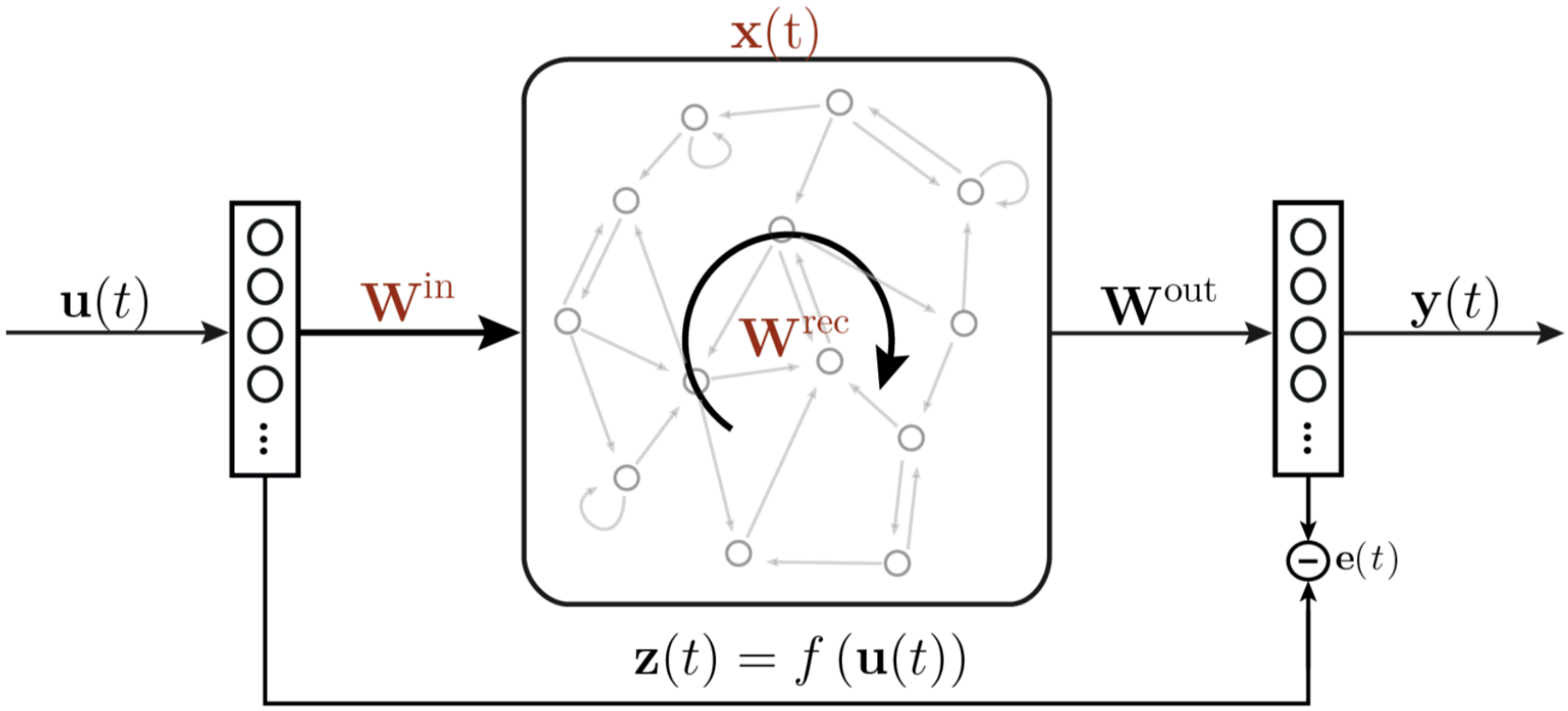
$$\boldsymbol{\Gamma} = \int d\vec{x}\, \vec{a}^{\mathsf{T}}\vec{a}$$

$$\boldsymbol{\Upsilon} = \int d\vec{x}\, \vec{a}^{\mathsf{T}}\vec{x}$$



Eliasmith, C. and Anderson, C. (2003). *Neural Engineering: Computation, Representation and Dynamics in neurobiological systems.* MIT Press.

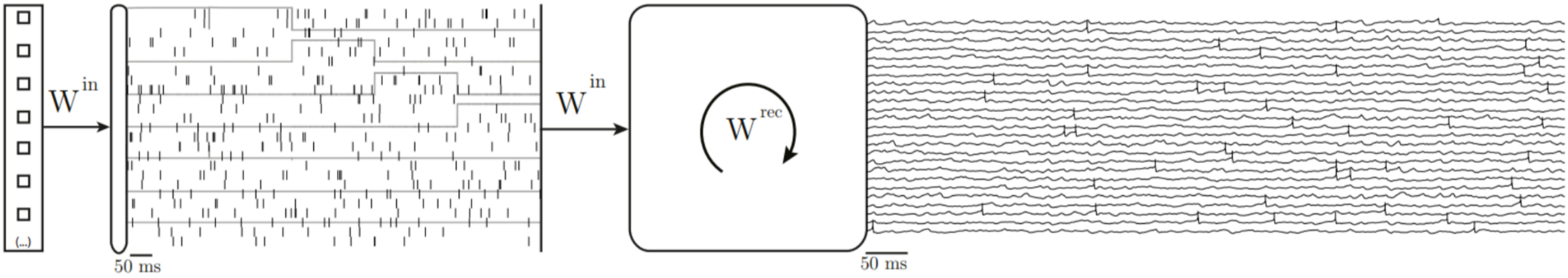
Exploration



- Specifications of circuit composition and dynamics (working examples will be provided)
  - neuron / synapse models
  - model parameters

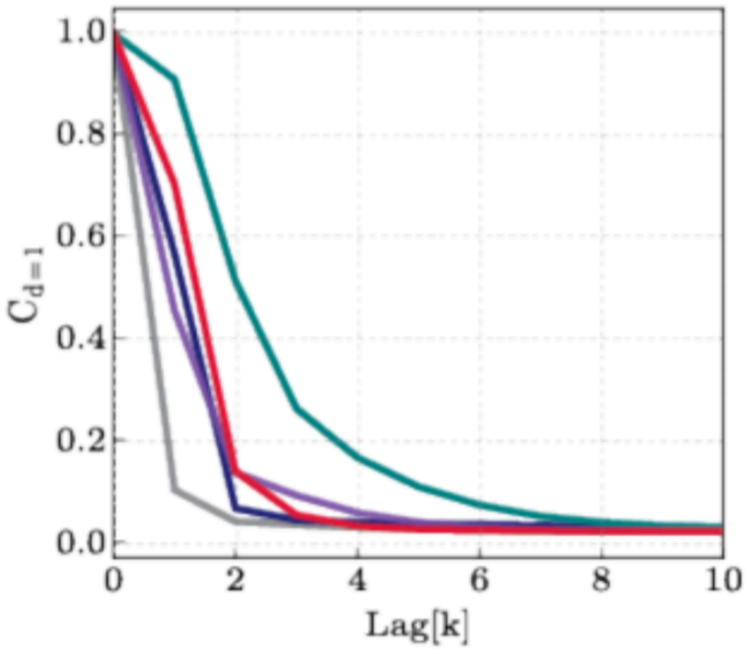
- Mappings (structure of input / recurrent connectivity)

Goals



Determine and compare features of the encoding and processing layers:

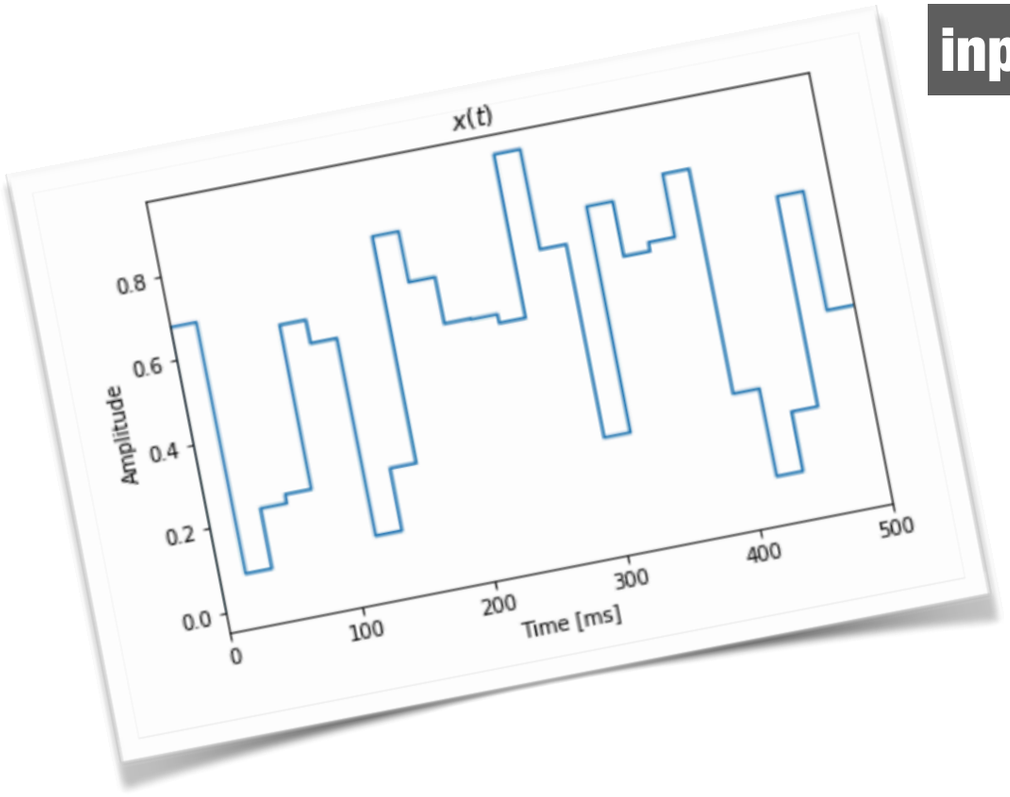
- Memory capacity, input sensitivity
- Activity statistics



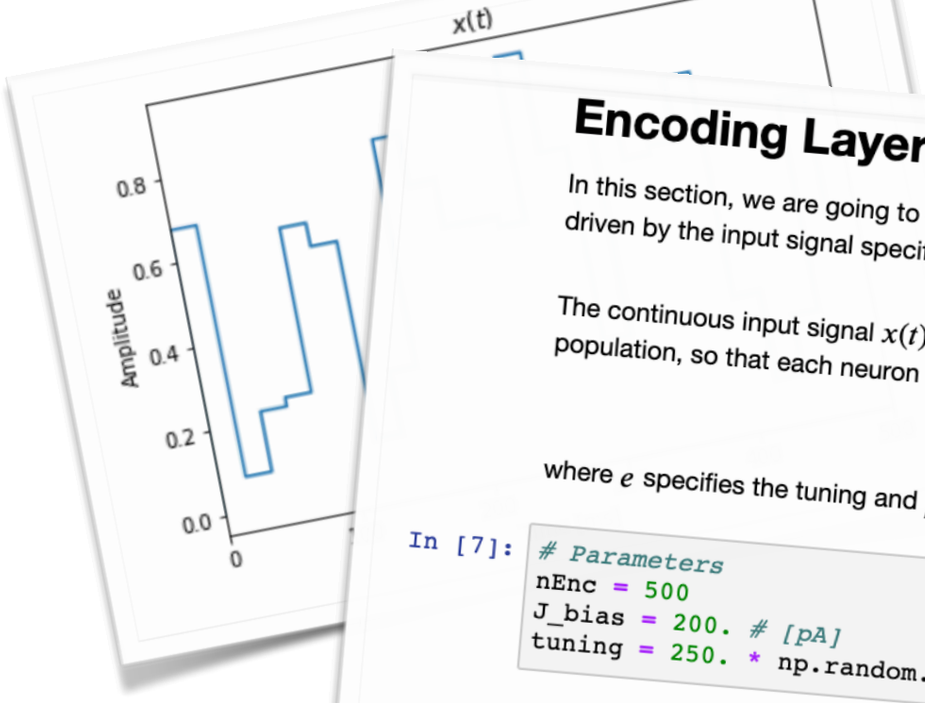
Explore the impact and consequences of different models, connectivity structures, etc.

pipeline

input → encoding → processing → readout → capacity



input → encoding → processing → readout → capacity



Encoding Layer

In this section, we are going to create a simple encoding layer, composed of a population of unconnected, passively stimulated neurons. These neurons are driven by the input signal specified before.

The continuous input signal $x(t)$ is converted to an input current $I_{in}(t)$ delivered to a population of spiking neurons. We randomize the tuning of this population, so that each neuron receives a slightly different "version" of the input signal, i.e.:

$$I_{in}(t) =< x(t)^T e > +\beta$$

where e specifies the tuning and β is a fixed bias current.

In [7]:
```
# Parameters
nEnc = 500
J_bias = 200. # [pA]
tuning = 250. * np.random.randn(nEnc) + 1000.
```

It is also important to randomize the initial states of the neurons as well as their firing thresholds (to introduce variability in the responses)

In [8]:
```
# randomize thresholds and initial states
thresholds = 5 * np.random.randn(nEnc) - 50.
Vm0 = np.array(np.random.uniform(low=-70., high=-50., size=int(nEnc)))
```

In [9]:
```
enc_layer = nest.Create('iaf_psc_delta', nEnc, {'I_e': J_bias})
```

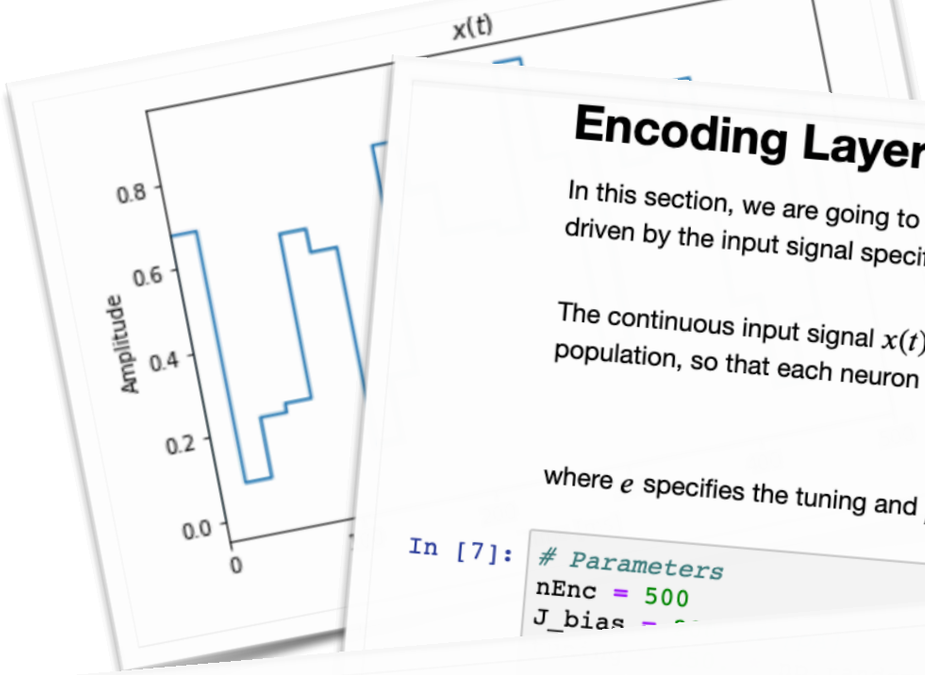
Now we create the input current generator and set the stimulus tuning and neuron parameters

In [10]:
```
step_generator = nest.Create('step_current_generator', nEnc)
amplitudes = np.zeros((nEnc, len(u)))
for n in range(nEnc):
    amplitudes[n, :] = u * tuning[n]
    nest.SetStatus([enc_layer[n]], {'V_m': Vm0[n], 'V_th': thresholds[n]})
    nest.SetStatus([step_generator[n]], {'amplitude_times': input_times,
                                         'amplitude_values': amplitudes[n]})
    nest.Connect([step_generator[n]], [enc_layer[n]])
```

Setup the recording devices (for now, we'll only record spikes, but later we can try to explore reading the V_m directly)

In [11]:
```
enc_spks = nest.Create('spike_detector')
nest.Connect(enc_layer, enc_spks)
```

input → encoding → processing → readout → capacity



Encoding Layer

In this section, we are going to create a simple encoding layer, composed of a population of unconnected, passively stimulated neurons. These neurons are driven by the input signal specified before.

The continuous input signal $x(t)$ is converted to an input current $I_{in}(t)$ delivered to a population of spiking neurons. We randomize the tuning of this population, so that each neuron receives a slightly different "version" of the input signal, i.e.:

$$I_{in}(t) =< x(t)^T e > +\beta$$

where e specifies the tuning and β is a fixed bias current.

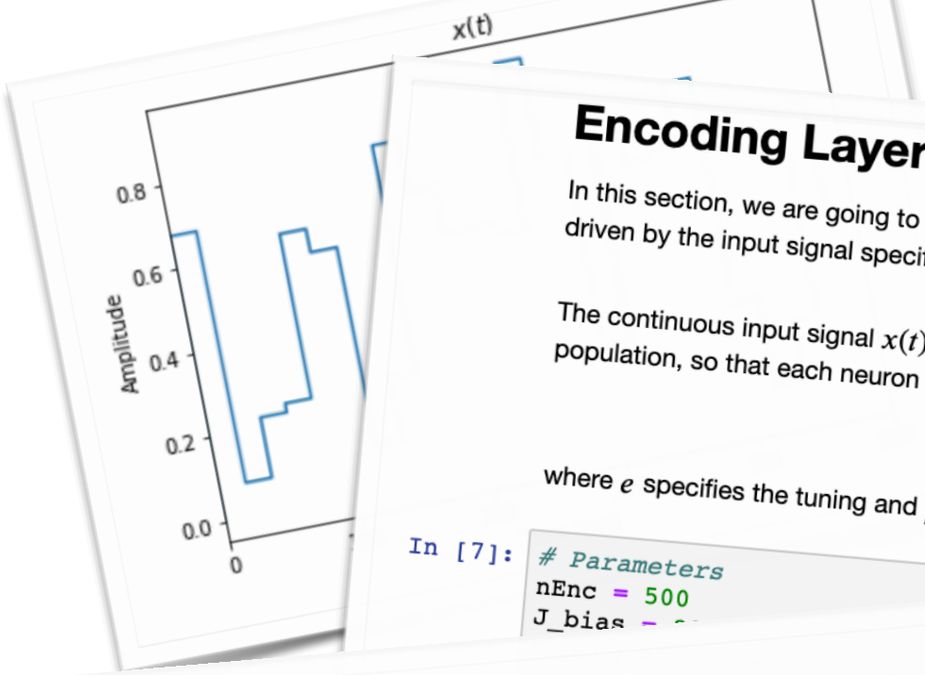
In [7]:
```
# Parameters
nEnc = 500
J_bias =
```

Processing Layer

The encoding layer is now fully specified, so the next step is to create the main network. For this example, we will use the standard balanced random network.

In [12]:
```
#### PARAMETERS ###
# network parameters
gamma = 0.25          # relative number of inhibitory connections
NE = 1000             # number of excitatory neurons (10.000 in [1])
NI = int(gamma * NE)  # number of inhibitory neurons
CE = 200              # indegree from excitatory neurons
CI = int(gamma * CE)  # indegree from inhibitory neurons

# synapse parameters
w = 0.1               # excitatory synaptic weight (mV)
g = 5.                # relative inhibitory to excitatory synaptic weight
d = 1.5               # synaptic transmission delay (ms)

# neuron paramters
neuron_params = {
    'C_m': 1.0,       # membrane capacity (pF)
    'E_L': 0.,        # resting membrane potential (mV)
    'I_e': 0.,        # external input current (pA)
    'V_m': 0.,        # membrane potential (mV)
    'V_reset': 10.,   # reset membrane potential after a spike (mV)
    'V_th': 20.,      # spike threshold (mV)
    't_ref': 2.0,     # refractory period (ms)
    'tau_m': 20.,     # membrane time constant (ms)
}
```

input → encoding → processing → readout → capacity



Encoding Layer

In this section, we are going to create a simple encoding layer, composed of a population of unconnected, passively stimulated neurons. These neurons are driven by the input signal specified before.

The continuous input signal $x(t)$ is converted to an input current $I_{in}(t)$ delivered to a population of spiking neurons. We randomize the tuning of this population, so that each neuron receives a slightly different "version" of the input signal, i.e.:

$$I_{in}(t) =< x(t)^T e > +\beta$$

where e specifies the tuning and β is a fixed bias current.

In [7]:
```
# Parameters
nEnc = 500
J_bias =
```

Processing Layer

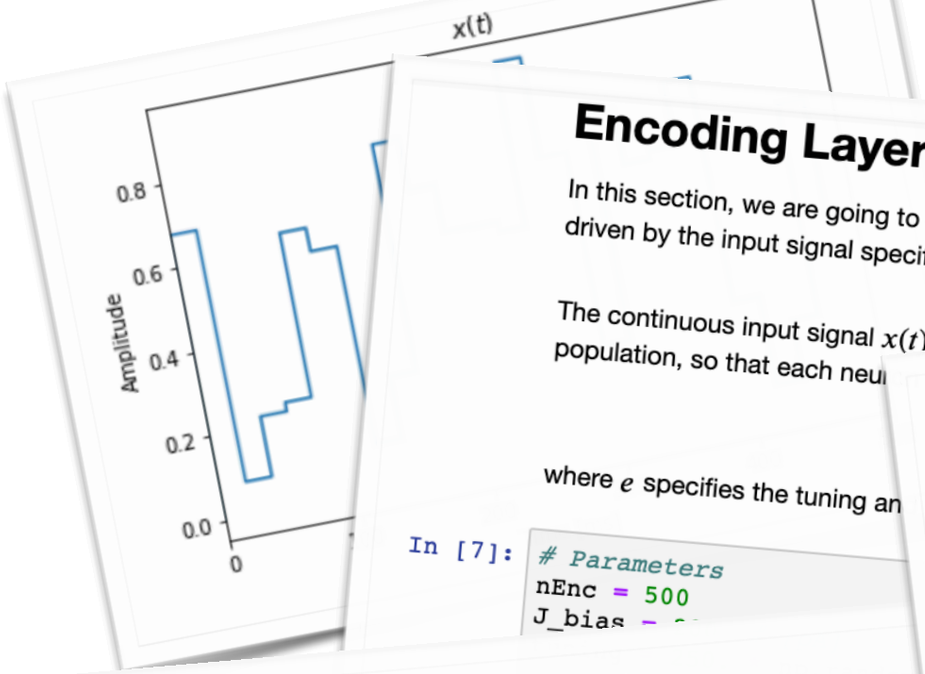
The encoding layer is now fully specified, so the next step is to create the main network. For this example, we will use the standard balanced random network.

In [12]:
```
#### PARAMETERS ###
# network parameters
gamma = 0.25                    # relative number of inhibitory connections
NE = 1000                       # number of excitatory neurons (10.000 in [1])
                                # number of inhibitory neurons
                                # degree from excitatory neurons
                                # degree from inhibitory neurons
    'tau_m' : 20.,
}
```

Create neurons and devices

In [13]:
```
# set default parameters for neurons and create neurons
nest.SetDefaults('iaf_psc_delta', neuron_params)
neurons_e = nest.Create('iaf_psc_delta', NE)
neurons_i = nest.Create('iaf_psc_delta', NI)

# create spike detectors
spikes_e = nest.Create('spike_detector')
nest.SetStatus(spikes_e, [{'withtime': True,
                           'withgid': True,
                           'to_file': False}])
```

input → encoding → processing → readout → capacity



Encoding Layer

In this section, we are going to create a simple encoding layer, composed of a popula
driven by the input signal specified before.

The continuous input signal $x(t)$ is
population, so that each neu

where e specifies the tuning an

In [7]:
```
# Parameters
nEnc = 500
J_bias =
```

Processing Layer

The encoding layer is now fully specified, so the next step is

In [12]:
```
#### PARAMETERS ###
# network parameters
gamma = 0.25
NE = 1000
                    # relative number
                    # number of excita
                    # number of inhibi
                    # degree from exc
    'tau_m' : 20.,
}
```

Create neurons and devices

In [13]:
```
# set default parameters for neurons
nest.SetDefaults('iaf_psc_delta', neu
neurons_e = nest.Create('iaf_psc_delta', NE)
neurons_i = nest.Create('iaf_psc_delta', NI)

# create spike detectors
spikes_e = nest.Create('spike_detector')
nest.SetStatus(spikes_e, [{'withtime': True,
                           'withgid': True,
                           'to_file': False}])
```

Connect network and devices

In [14]:
```
# E synapses
syn_exc = {'delay': d, 'weight': w}
conn_exc = {'rule': 'fixed_indegree', 'indegree': CE}
nest.Connect(neurons_e, neurons_e, conn_exc, syn_exc)
nest.Connect(neurons_e, neurons_i, conn_exc, syn_exc)

# I synapses
syn_inh = {'delay': d, 'weight': - g * w}
conn_inh = {'rule': 'fixed_indegree', 'indegree': CI}
nest.Connect(neurons_i, neurons_e, conn_inh, syn_inh)
nest.Connect(neurons_i, neurons_i, conn_inh, syn_inh)

# spike detector
nest.Connect(neurons_e, spikes_e)
```

Connect encoding layer to main circuit

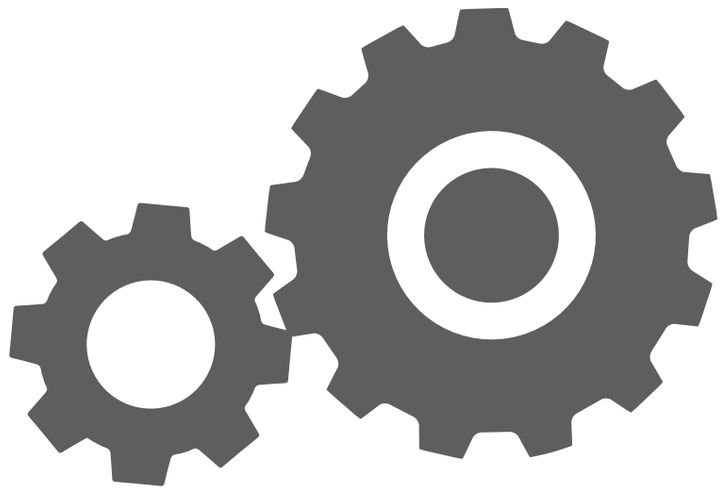
For this example, we will use a normal excitatory synapse to connect the encoders to both E and I neurons

In [15]:
```
nest.Connect(enc_layer, neurons_e, conn_exc, syn_exc)
nest.Connect(enc_layer, neurons_i, conn_exc, syn_exc)
```

input → encoding → processing → readout → capacity

Simulate and analyse

```
In [16]: nest.Simulate(T*duration)
```



Decoding

If the input-driven dynamics contains enough information, we should be able to reconstruct some target signal.. For this purpose, we will use standard linear regression (see functions). If the state of the circuit (be it the encoding layer or the main processing layer is stored in a matrix A and the target signal to reconstruct is y, we can estimate the optimal linear projection as:

$$D = \left(A^T A\right)^{-1} A^T Y$$

In the simplest case, we want to reconstruct the original signal, i.e. $y(t) = x(t)$.

The circuit's capacity to reconstruct $y(t) = f(x(t))$ by linearly combining the population responses is given by:

$$\mathrm{C[A, y]} = \frac{\left(A^T A\right)^{-1} Ay}{\|y\|^2}$$

where the target function $y(t)$ can be any arbitrary function of the input, i.e. $y(t) = f(x(t))$.

We begin by evaluating how well how well the input signal $x(t)$ is encoded in the activity of the encoding layer $A_{enc}(t)$ and how this is transferred to the main processing circuit $A_{proc}(t)$.

In [18]:
```
def compute_capacity(a, y):
    """
    Compute capacity to reconstruct y based on linearly combining a
    :param a: state matrix (NxT)
    :param y: target output (1xT)
    :return y_hat: estimated signal
     :return capacity:
     :return error:
    """
    reg = LinearRegression(n_jobs=-1, fit_intercept=False, normalize=True, copy_X=False).fit(a.T, y)
    y_hat = np.dot(reg.coef_, a)
    error = np.mean((y - y_hat) ** 2)
    capacity = 1. - (error/np.var(y))
    return y_hat, capacity, error

enc_estimate, enc_capacity, enc_error = compute_capacity(enc_states, signal)
circ_estimate, circ_capacity, circ_error = compute_capacity(e_states, signal)

print("Encoding Layer: \n\t- Capacity={0}, MSE={1}".format(str(enc_capacity), str(enc_error)))
print("Main Circuit: \n\t- Capacity={0}, MSE={1}".format(str(circ_capacity), str(circ_error)))
```

Decoding

If the input-driven dynamics contains enough information, we should be able to reconstruct some target signal.. For this purpose, we will use standard linear regression (see functions). If the state of the circuit (be it the encoding layer or the main processing layer is stored in a matrix A and the target signal to reconstruct is y, we can estimate the optimal linear projection as:

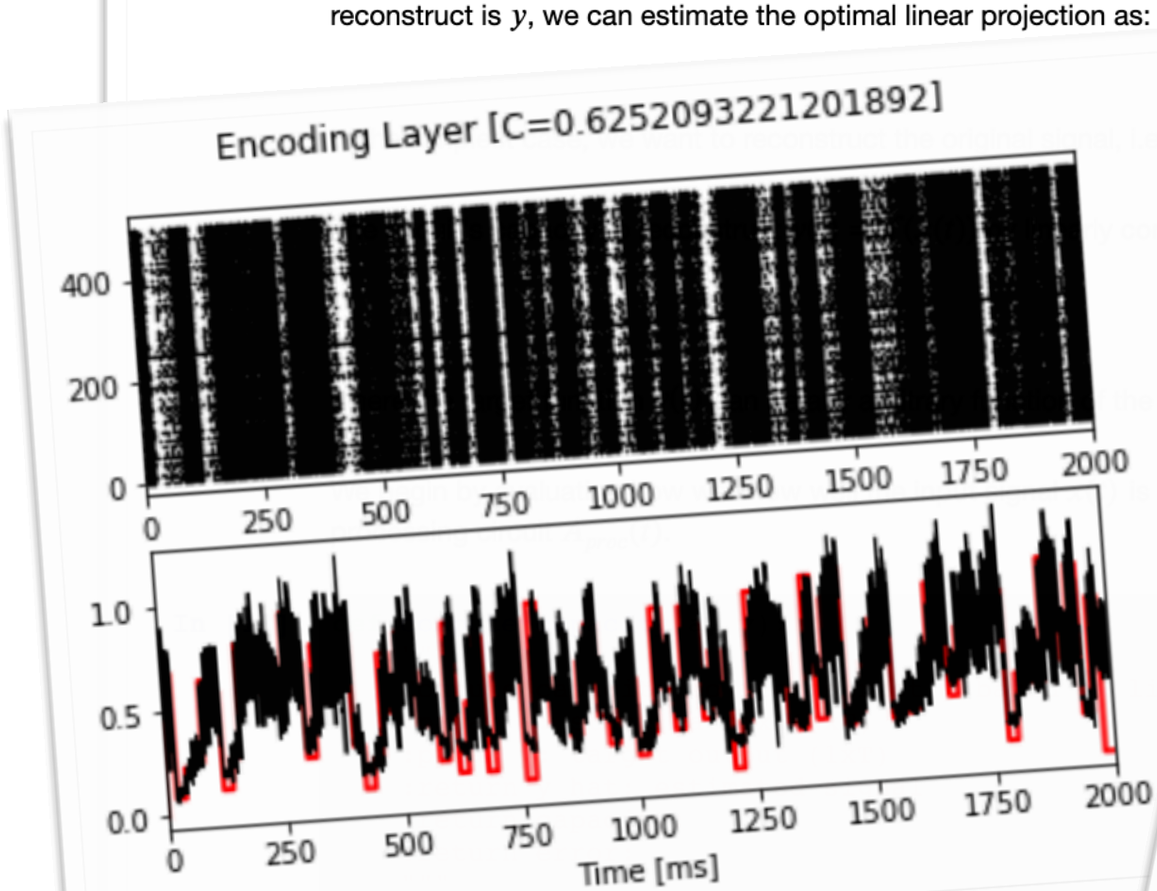
$$\beta = \left(A^T A\right)^{-1} A^T Y$$

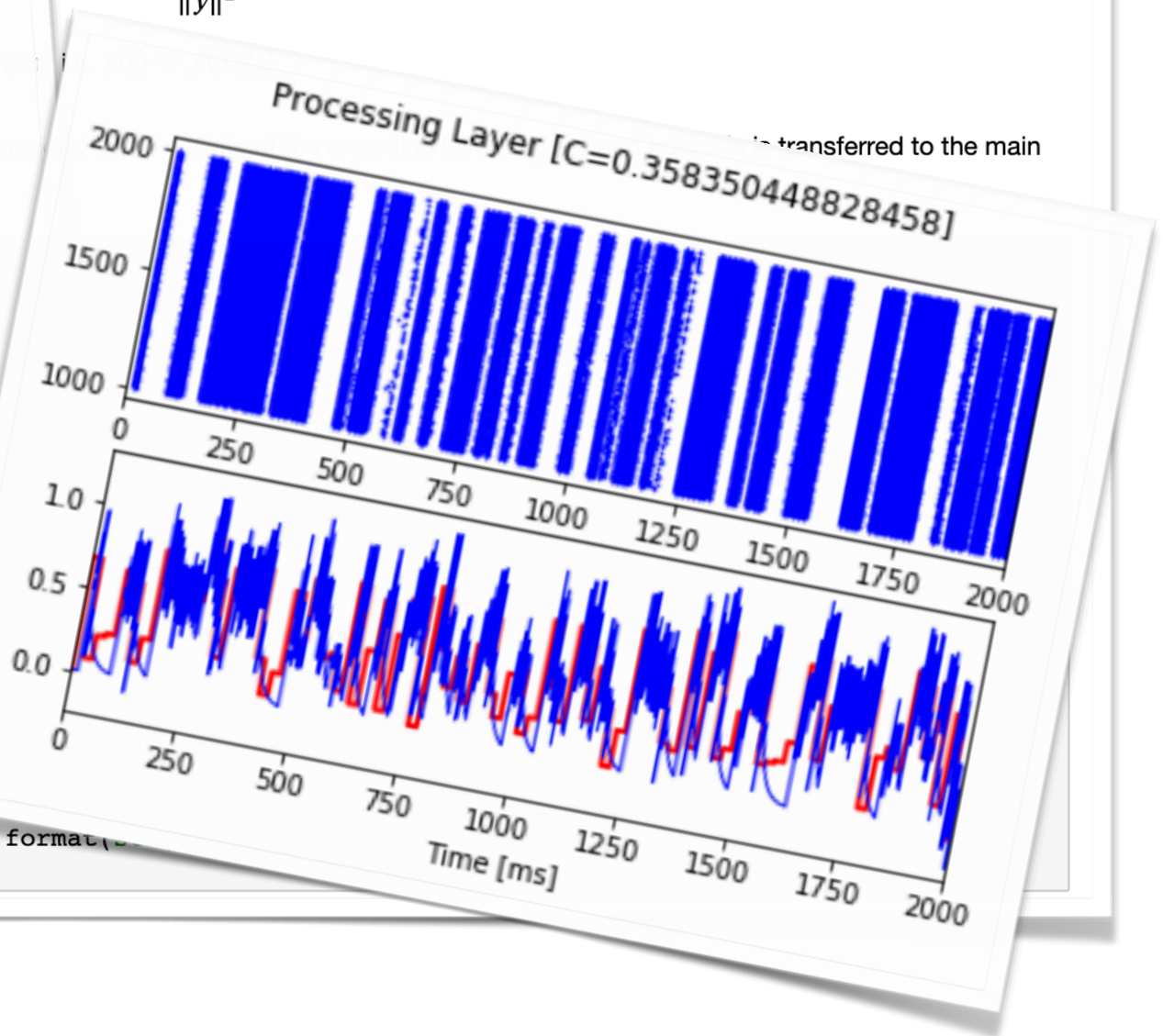
$$y(t) = x(t).$$

ng the population responses is given by:

$$\beta[x, y] = \frac{\left(A^T A\right)^{-1} Ay}{\|y\|^2}$$

is transferred to the main






```
y_hat = np.dot(reg.coef_, a)
error = np.mean((y - y_hat) ** 2)
capacity = 1. - (error/np.var(y))
return y_hat, capacity, error

enc_estimate, enc_capacity, enc_error = comput
circ_estimate, circ_capacity, circ_error = co

print("Encoding Layer: \n\t- Capacity={0}, MSE-{
print("Main Circuit: \n\t- Capacity={0}, MSE={1}".format(
```

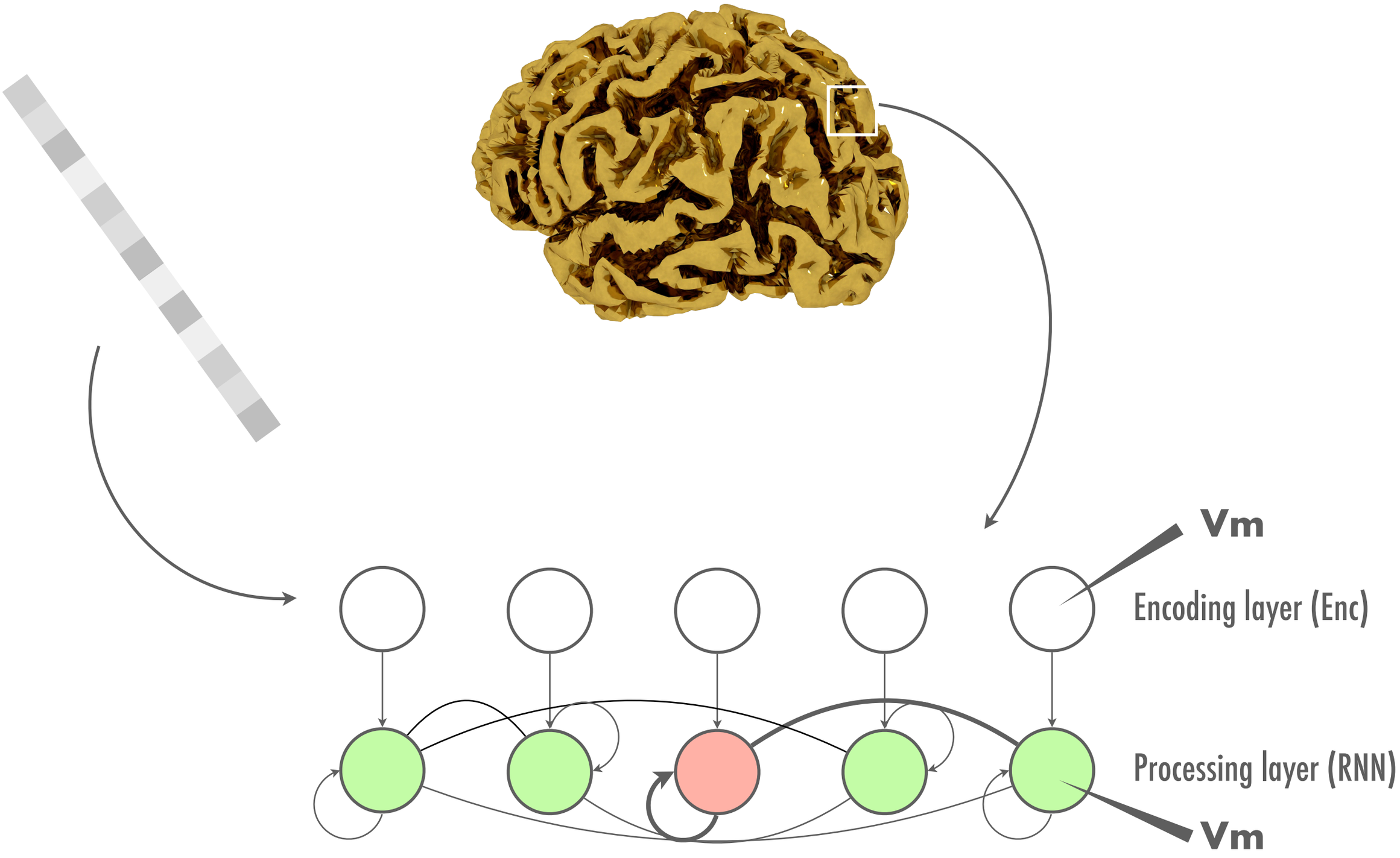
Memory capacity

So far, we have measured the ability to reconstruct the input signal $x(t)$, using the population responses $A(t) = G[x(t), a(t)]$. The complexity of the nonlinear functional G, determined by the neuronal dynamics and synaptic interactions may allow the circuit to retain information about the input for some time (fading memory). So, now we evaluate the circuit's memory capacity, by determining the ability to use the current population state $A(t)$ to reconstruct the input, at various time lags, i.e. we set the target functions to be the original input shifted in time. The idea being that if the circuit has memory, the present state at any given time t should contain information about the input that is driving it $x(t)$, but also about past history of the input (fading memory), i.e. $x(t - \tau_{\text{lag}})$, for $\tau_{\text{lag}} \in [0, \tau_{\text{lag}}^{\max}]$.

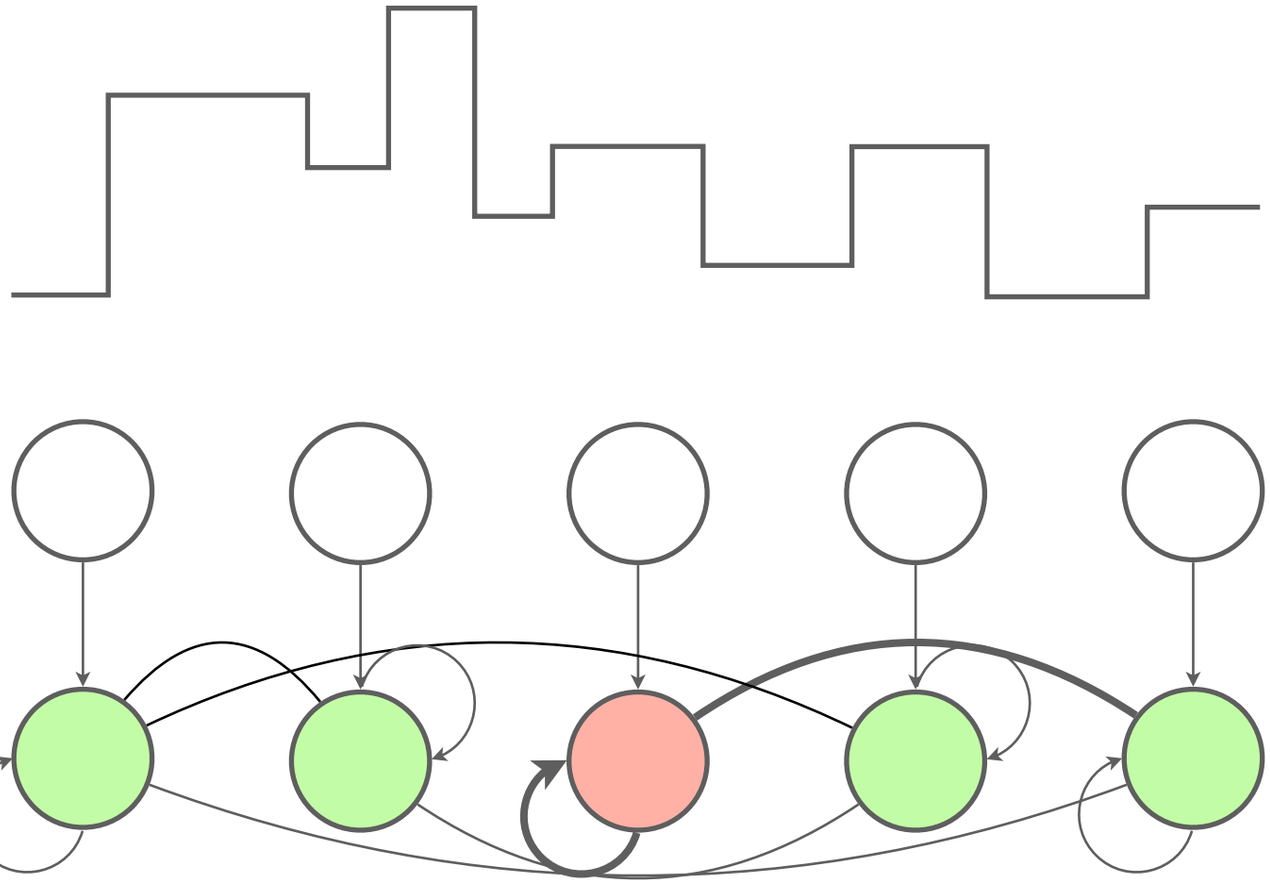
In [22]:
```
max_lag = 100.  # [ms] in this example
step_lag = 10.  # [ms] - if != dt (index the time axis)
time_lags = np.arange(0., max_lag, step_lag)
indices = [np.where(idx == time_vector)[0][0] for idx in time_lags]

encoder_capacity = []
circuit_capacity = []

for idx, lag in zip(indices, time_lags):

    # shift the target signal
    if idx > 0:
        shifted_signal = signal[:-idx]
    else:
        shifted_signal = signal

    # shift the population states
    enc_st = enc_states[:, idx:]
    circ_st = e_states[:, idx:]

    # compute capacity
    enc_estimate, enc_capacity, enc_error = compute_capacity(enc_st, shifted_signal)
    circ_estimate, circ_capacity, circ_error = compute_capacity(circ_st, shifted_signal)

    print("Lag = {0} ms".format(str(lag)))
    print("Encoding Layer: \n\t- Capacity={0}, MSE={1}".format(str(enc_capacity), str(enc_error)))
    print("Main Circuit: \n\t- Capacity={0}, MSE={1}".format(str(circ_capacity), str(circ_error)))

    encoder_capacity.append(enc_capacity)
    circuit_capacity.append(circ_capacity)

fig = pl.figure()
ax1 = fig.add_subplot(121)
ax2 = fig.add_subplot(122)

ax1.plot(time_lags, encoder_capacity)
ax2.plot(time_lags, circuit_capacity)

pl.show()

print("Total capacity (encoder): {0} ms".format(str(np.sum(encoder_capacity)*step_lag)))
print("Total capacity (processor): {0} ms".format(str(np.sum(circuit_capacity)*step_lag)))
```

Tommy's part

input and encoding


Vm
Encoding layer (Enc)
Processing layer (RNN)
Vm

input

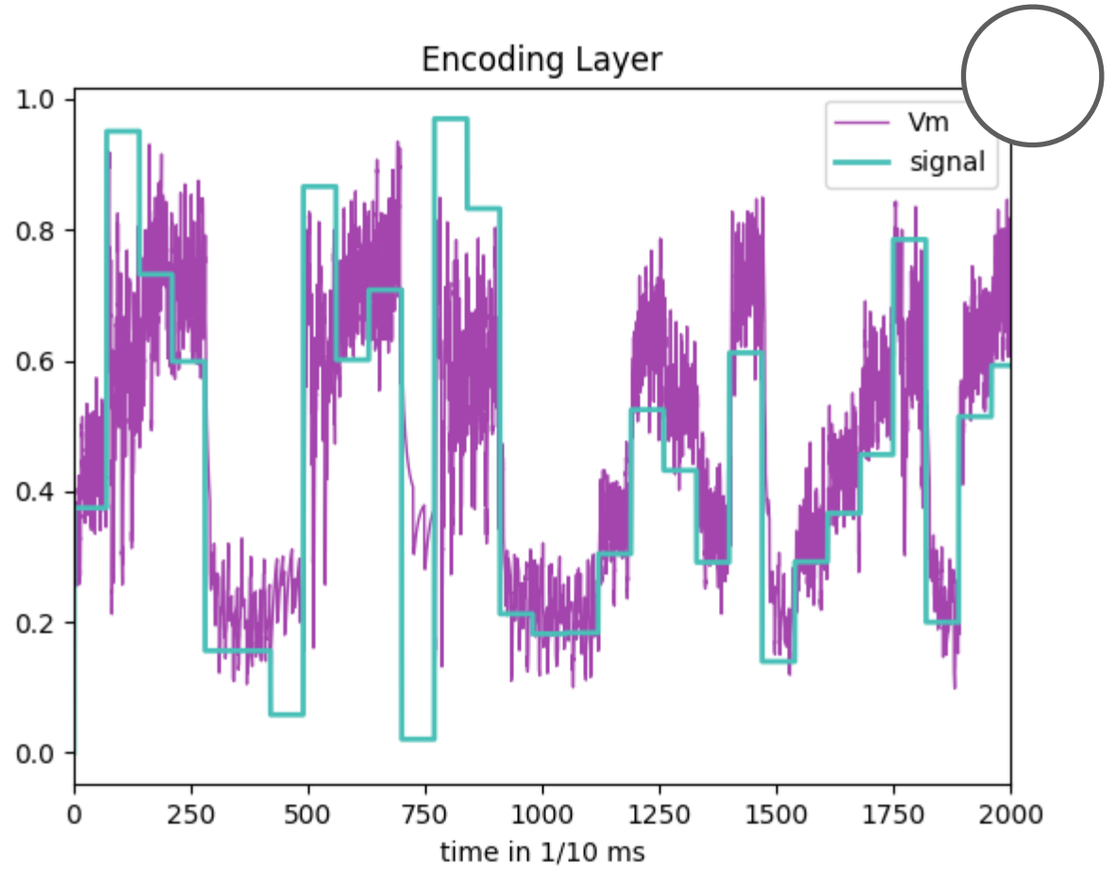


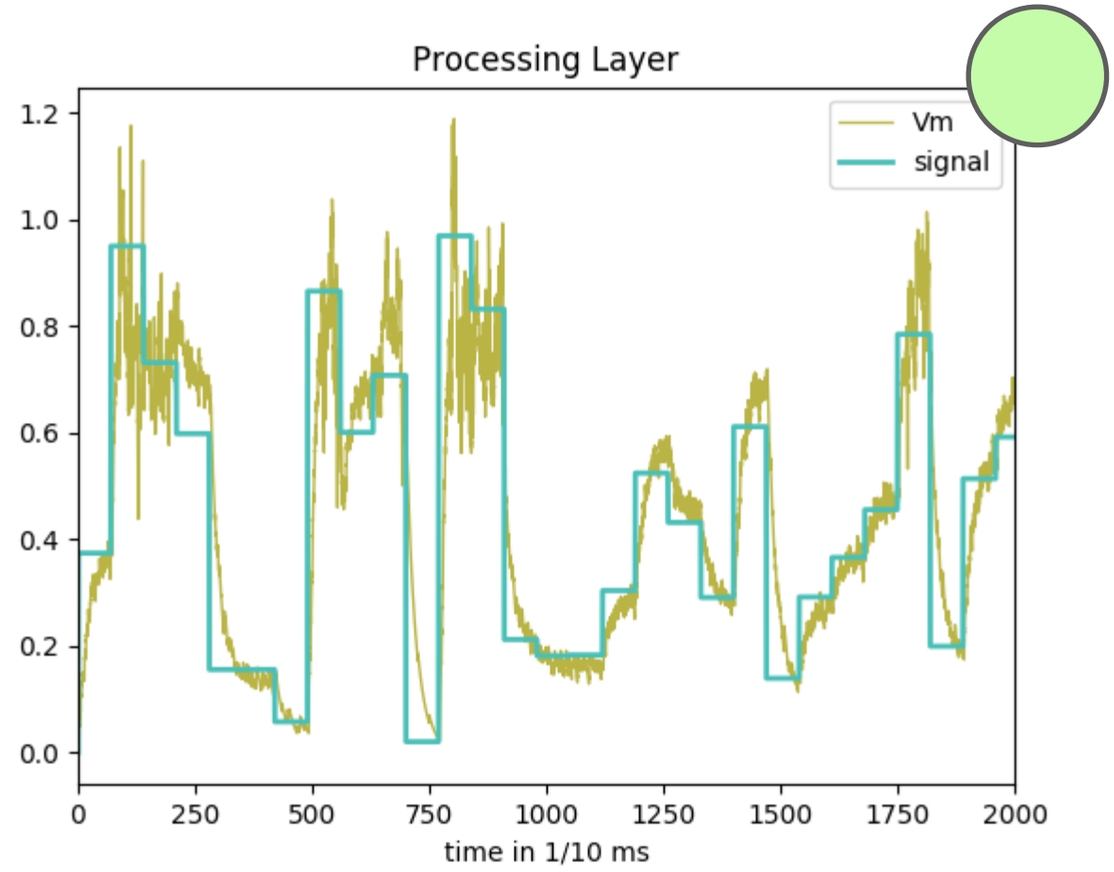
100 neurons

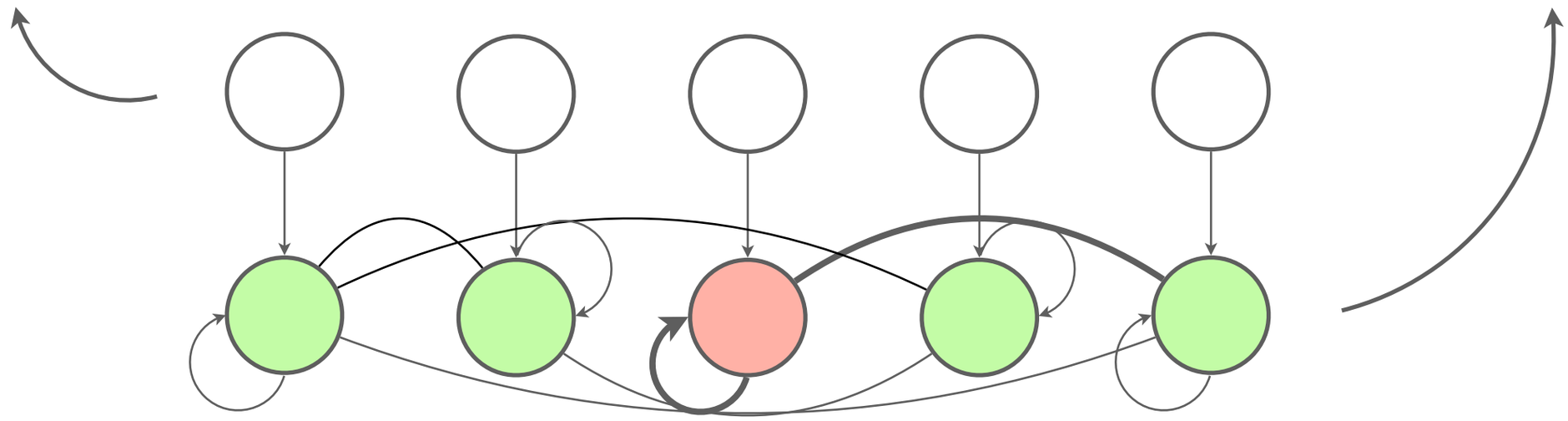
500 neurons

125 neurons

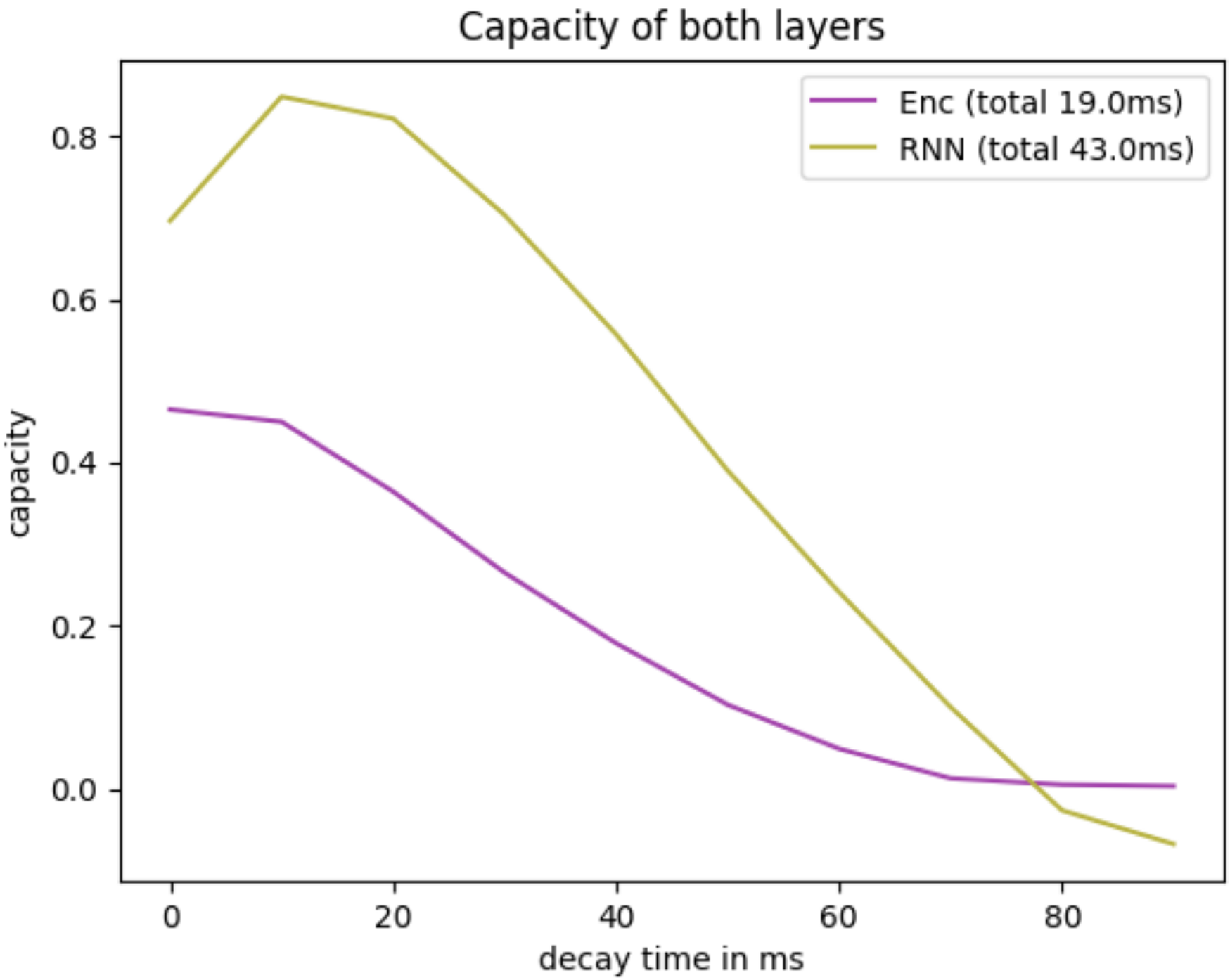
example


Encoding Layer
Vm
signal
1.0
0.8
0.6
0.4
0.2
0.0
0
250
500
750
1000
1250
1500
1750
2000
time in 1/10 ms


Processing Layer
Vm
signal
1.2
1.0
0.8
0.6
0.4
0.2
0.0
0
250
500
750
1000
1250
1500
1750
2000
time in 1/10 ms

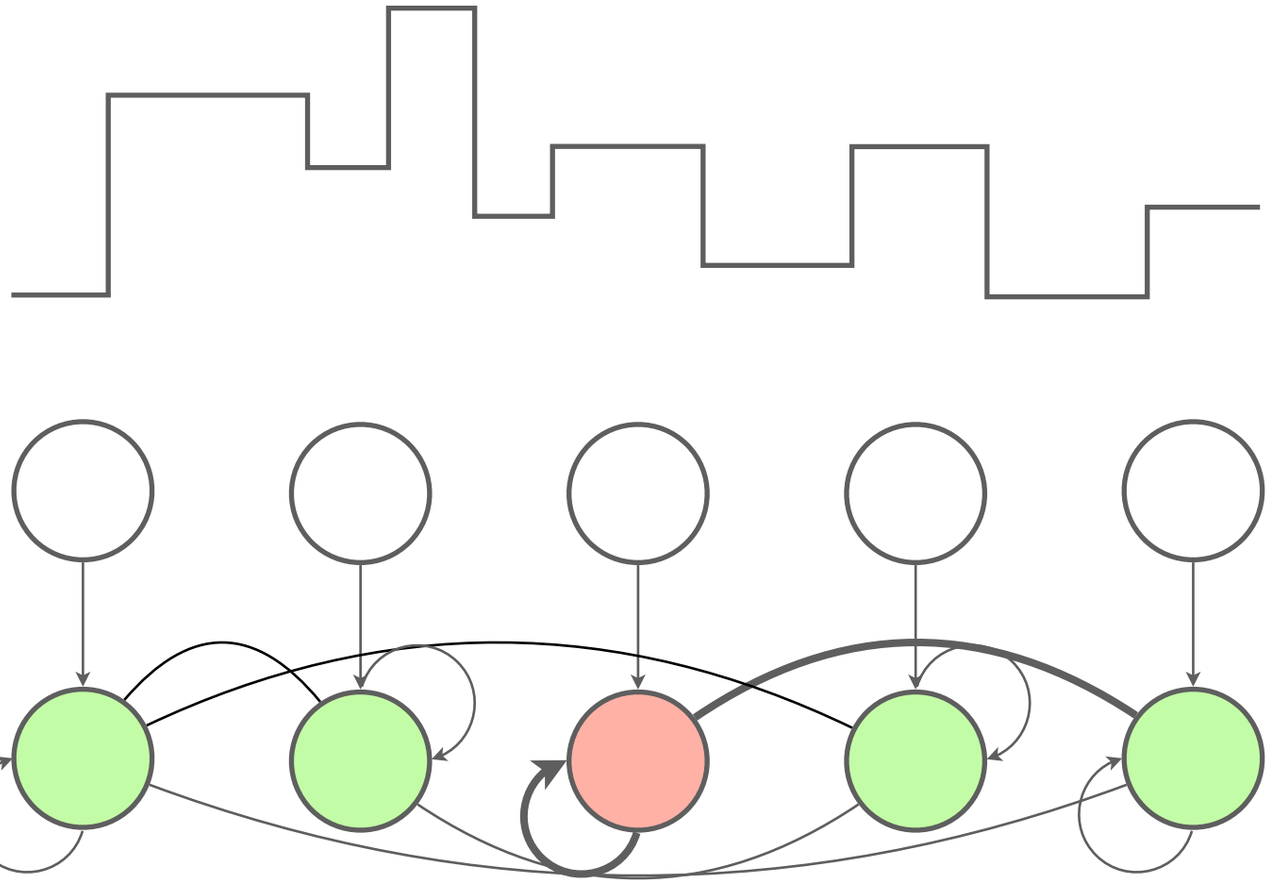


example


Capacity of both layers
Enc (total 19.0ms)
RNN (total 43.0ms)
capacity
0.8
0.6
0.4
0.2
0.0
0
20
40
60
80
decay time in ms

example

input

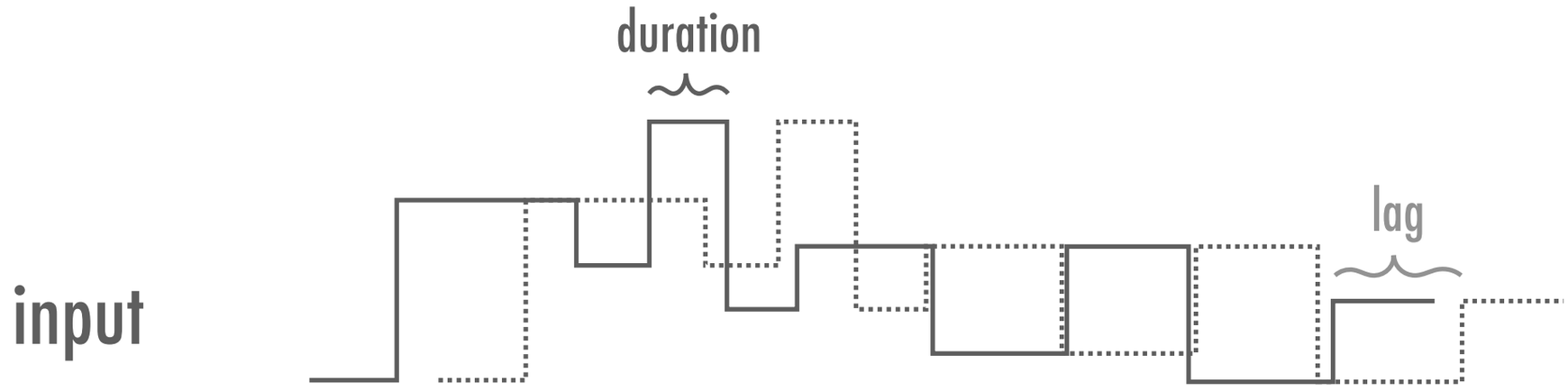


enc neurons

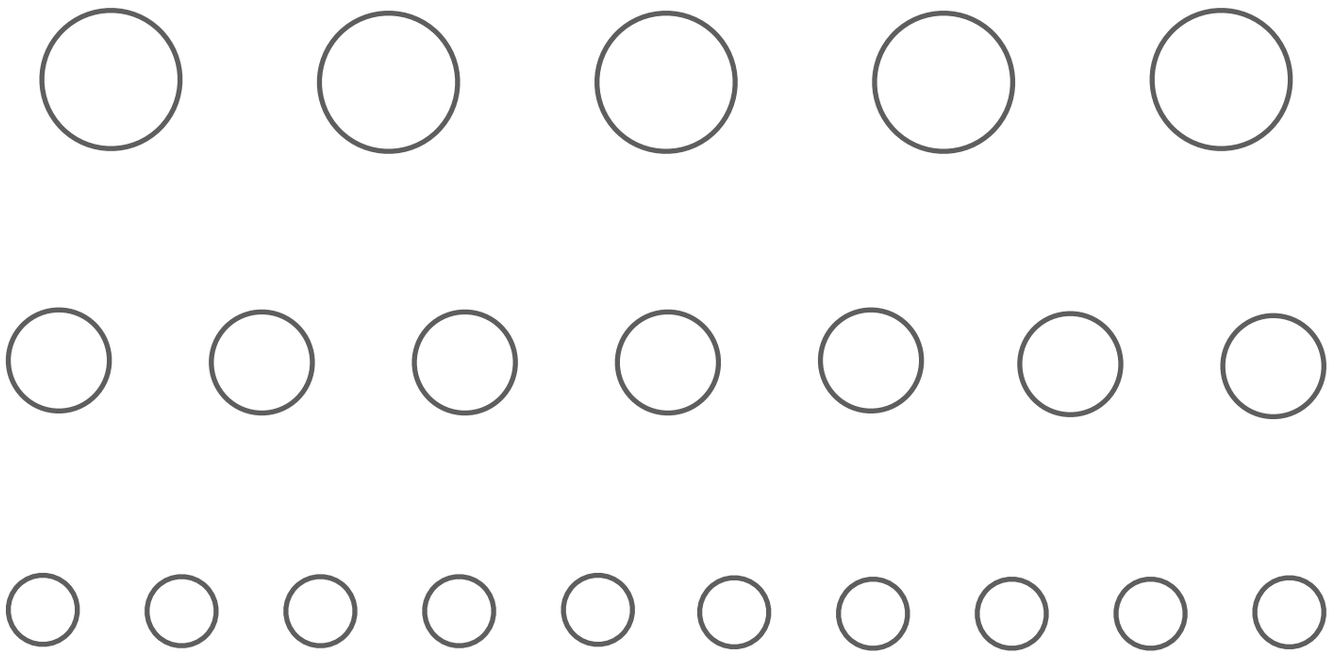
RNN ex neurons

RNN in neurons

parameters

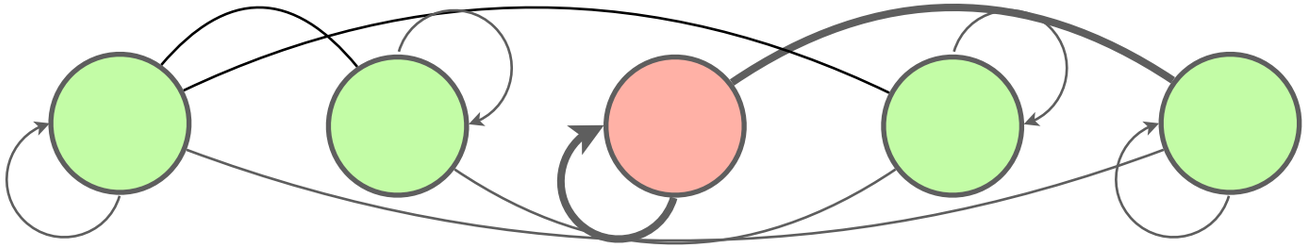


parameters

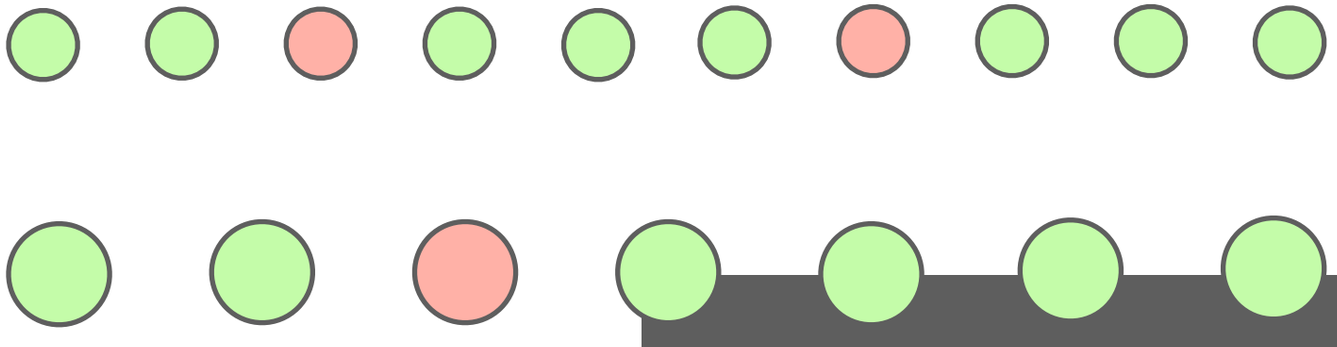


parameters

RNN ex neurons



RNN in neurons



parameters

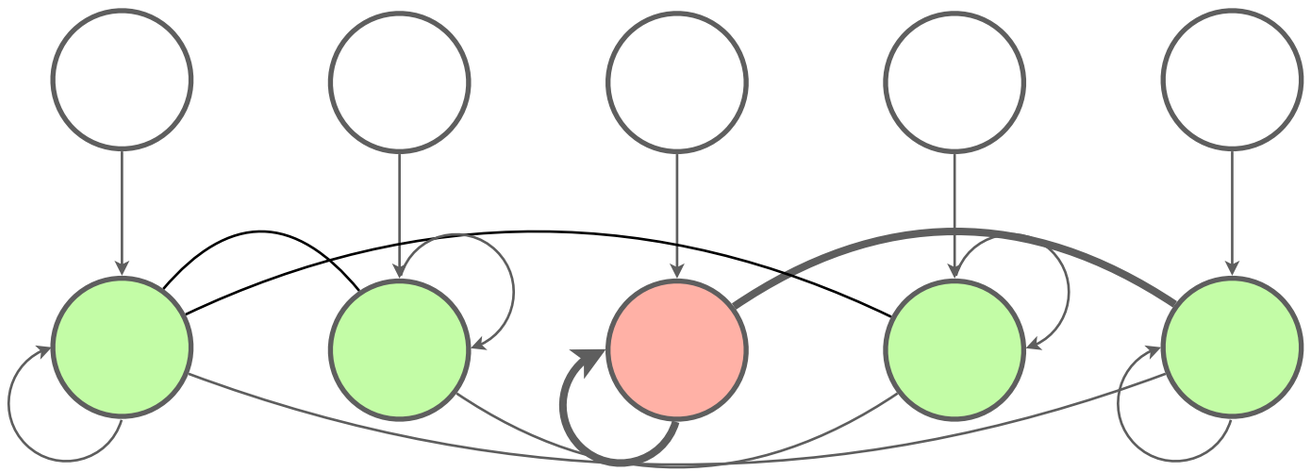
Network scaling (NS):

NS = # enc neurons / 100

NS = # RNN ex neurons / 500

NS = RNN ex in-degree / 100

enc neurons



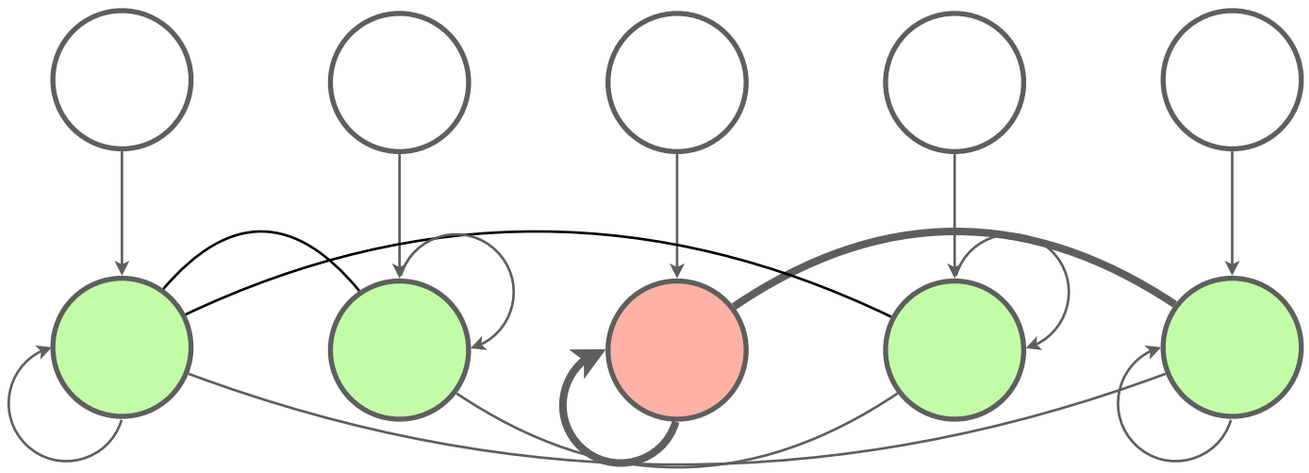
RNN ex neurons

RNN in neurons

remark on scaling

NS 1

100 neurons



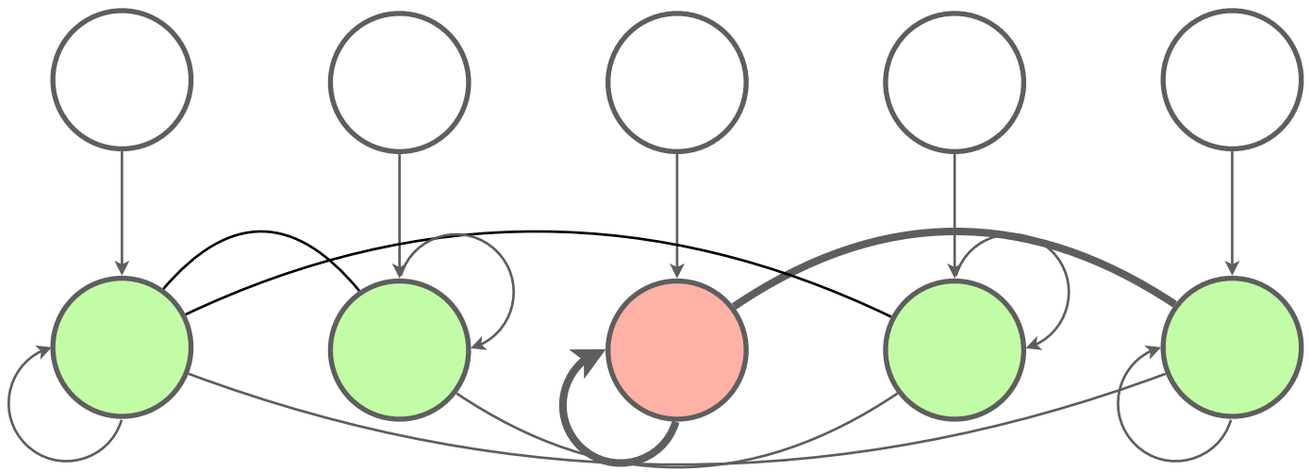
500 neurons

125 neurons

remark on scaling

NS 10

1000 neurons

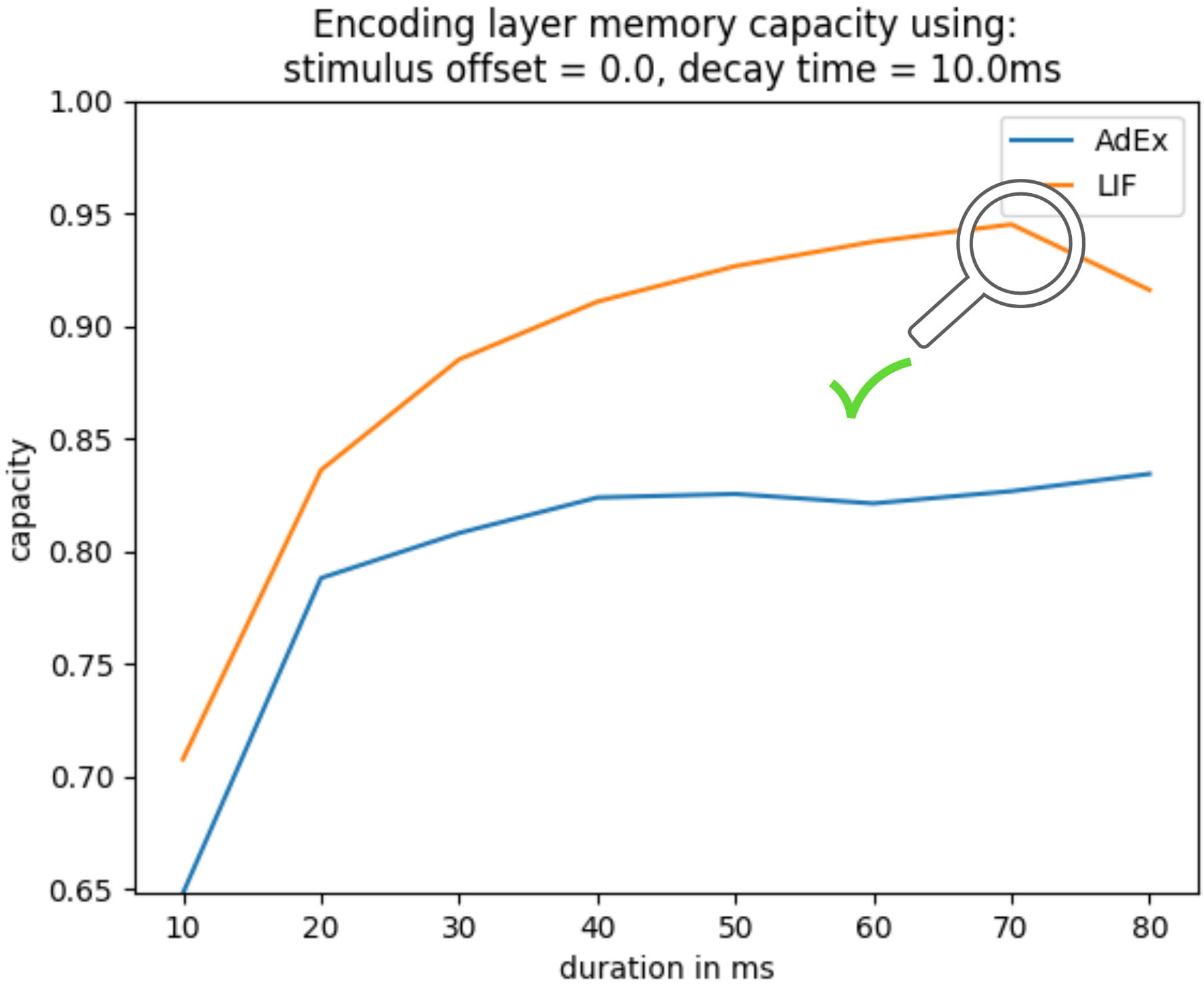


5000 neurons

1250 neurons

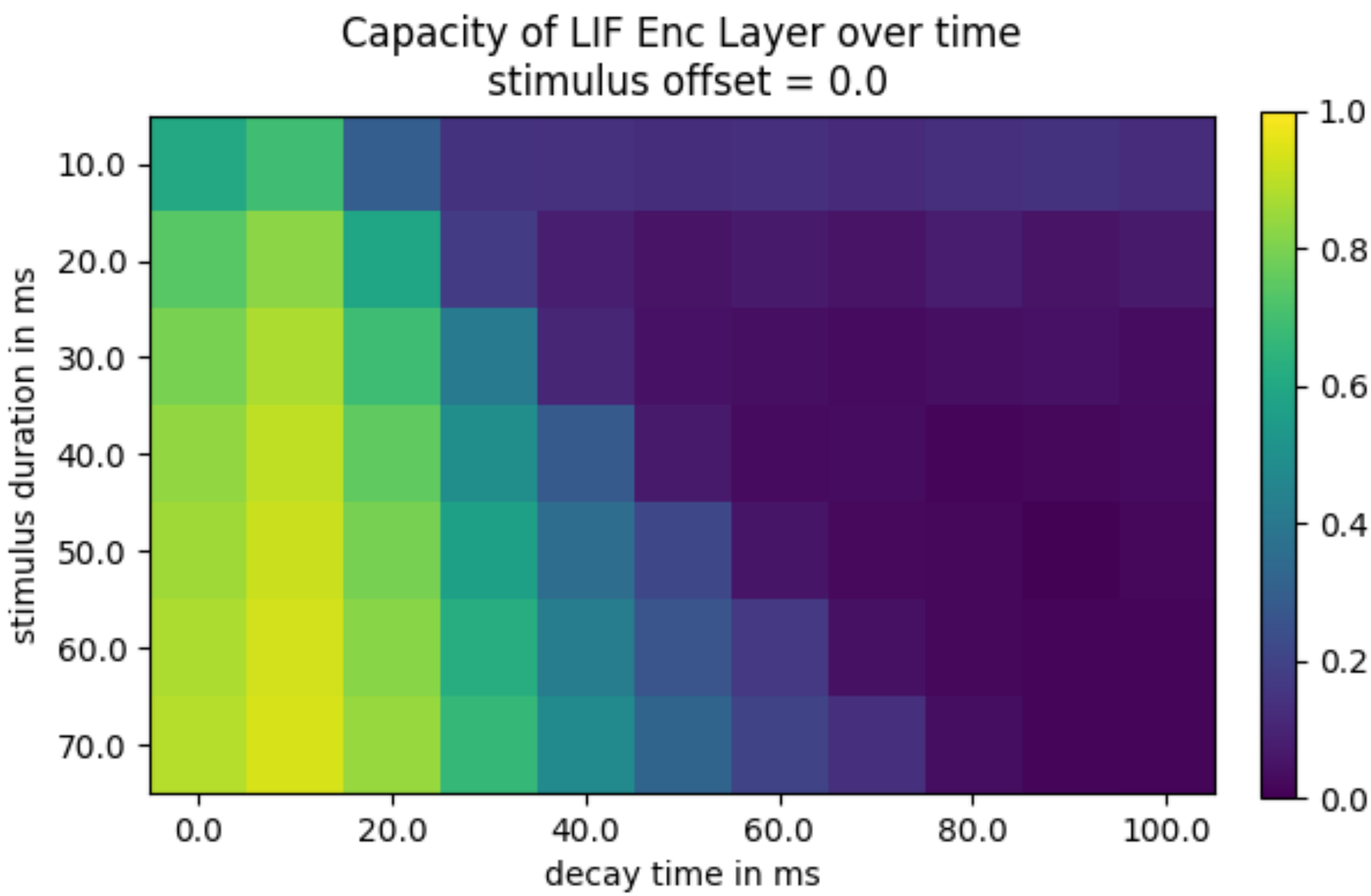
remark on scaling

NS 1


Encoding layer memory capacity using:
stimulus offset = 0.0, decay time = 10.0ms
AdEx
LIF
capacity
1.00
0.95
0.90
0.85
0.80
0.75
0.70
0.65
10
20
30
40
50
60
70
80
duration in ms

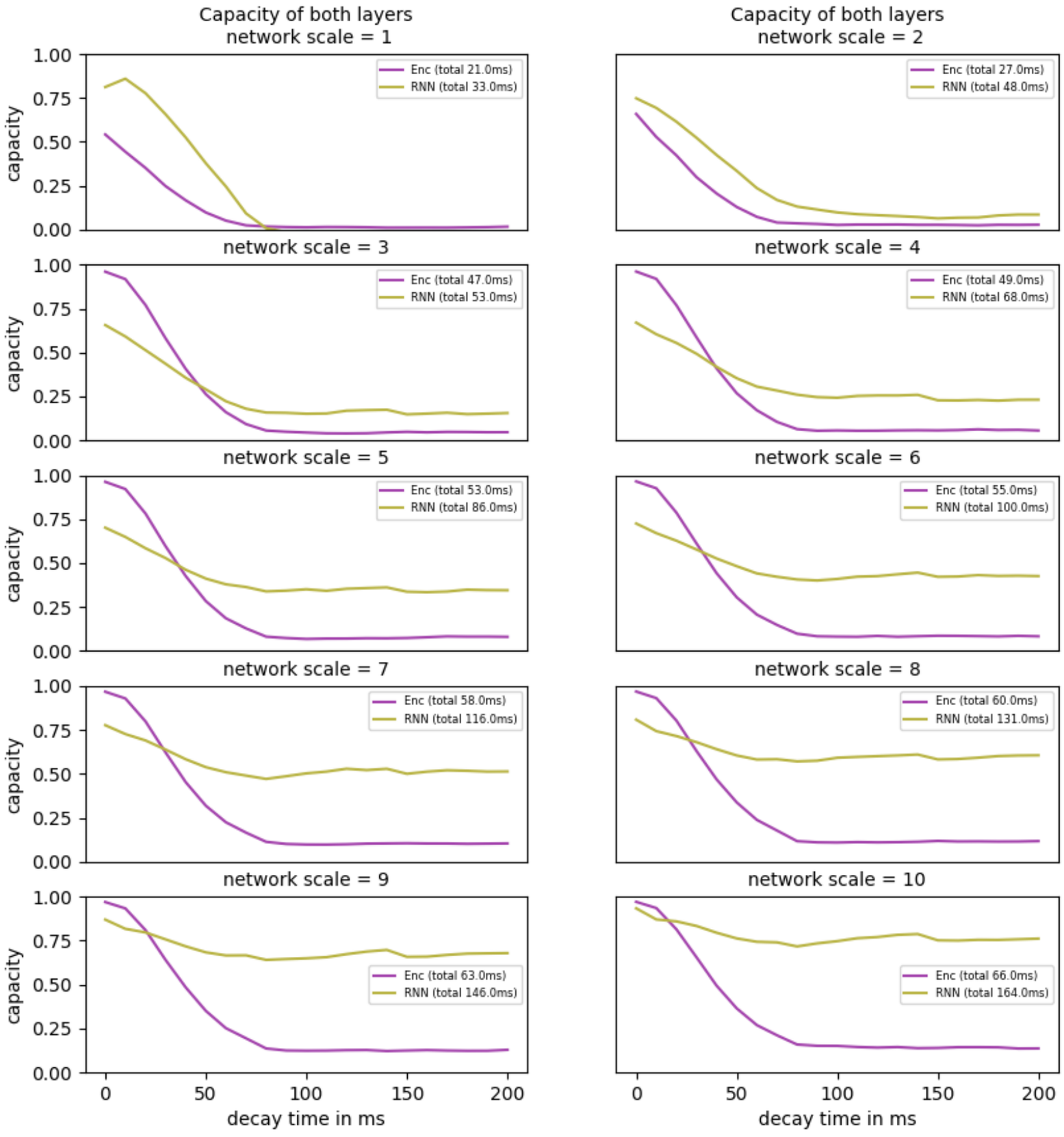
results

NS 1


Capacity of LIF Enc Layer over time
stimulus offset = 0.0
stimulus duration in ms
10.0
20.0
30.0
40.0
50.0
60.0
70.0
0.0
20.0
40.0
60.0
80.0
100.0
decay time in ms
1.0
0.8
0.6
0.4
0.2
0.0

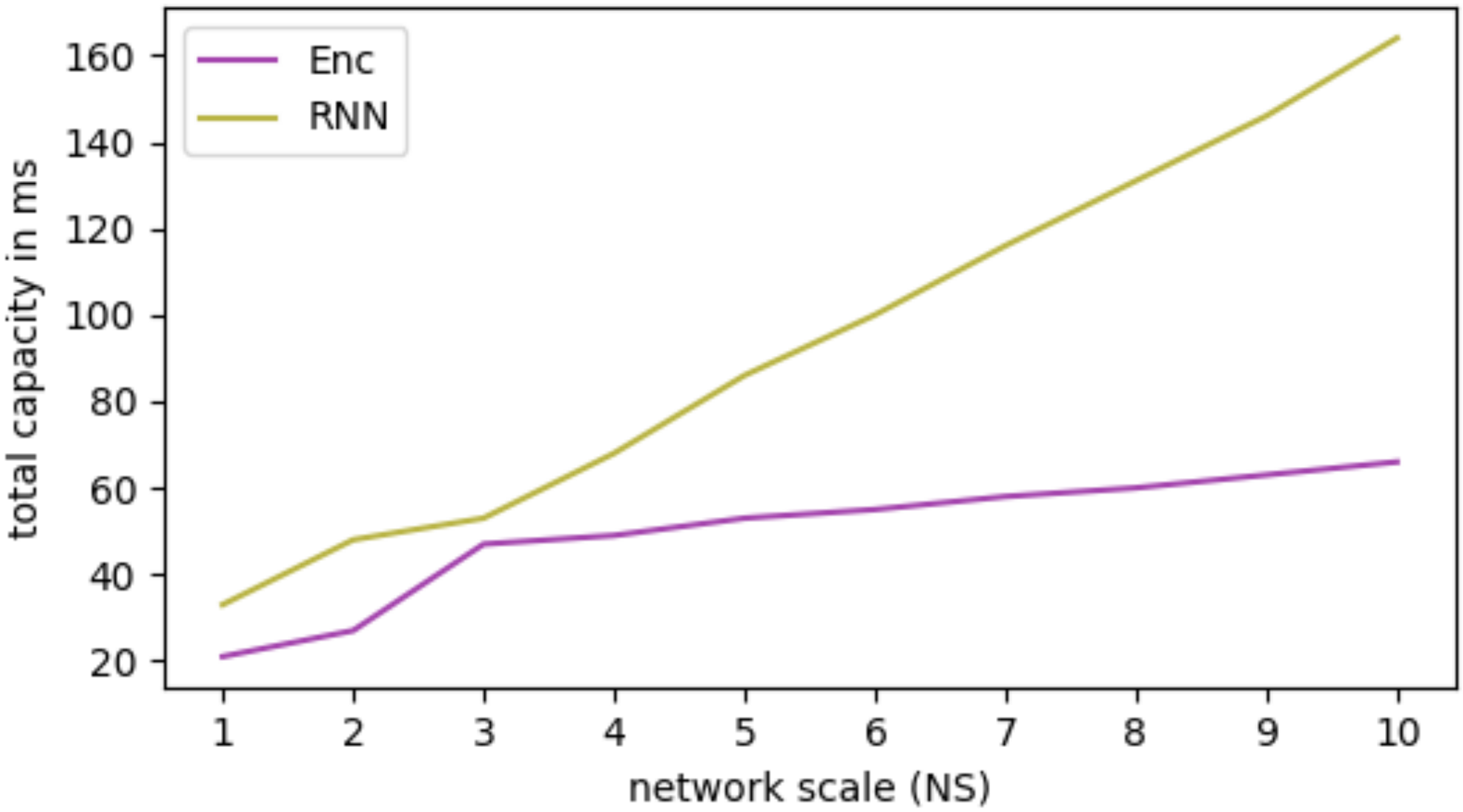
results

NS 1-10



results

NS 1-10

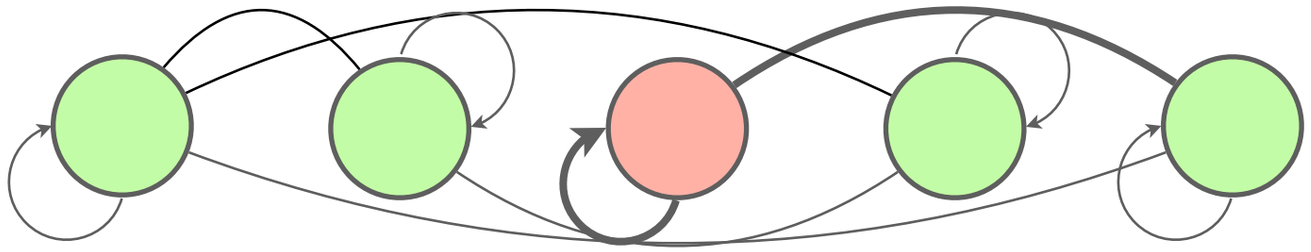


results

Mathias' part

processing layer

RNN ex neurons



RNN in neurons

Processing Layer

The encoding layer is now fully specified, so the next step is to create the main network. For this example, we will use the standard balanced random network.

In [12]:
```
#### PARAMETERS ###
# network parameters
gamma = 0.25                    # relative number of inhibitory connections
NE = 1000                       # number of excitatory neurons (10.000 in [1])
NI = int(gamma * NE)            # number of inhibitory neurons
CE = 200                        # indegree from excitatory neurons
CI = int(gamma * CE)            # indegree from inhibitory neurons

# synapse parameters
w = 0.1                         # excitatory synaptic weight (mV)
g = 5.                          # relative inhibitory to excitatory synaptic weight
d = 1.5                         # synaptic transmission delay (ms)

# neuron paramters
neuron_params = {               # membrane capacity (pF)
    'C_m': 1.0,                 # resting membrane potential (mV)
    'E_L': 0.,                  # external input current (pA)
    'I_e': 0.,                  # membrane potential (mV)
    'V_m': 0.,                  # reset membrane potential after a spike (mV)
    'V_reset': 10.,             # spike threshold (mV)
    'V_th': 20.,                # refractory period (ms)
    't_ref': 2.0,               # membrane time constant (ms)
    'tau_m': 20.,
}
```

w excitatory weights

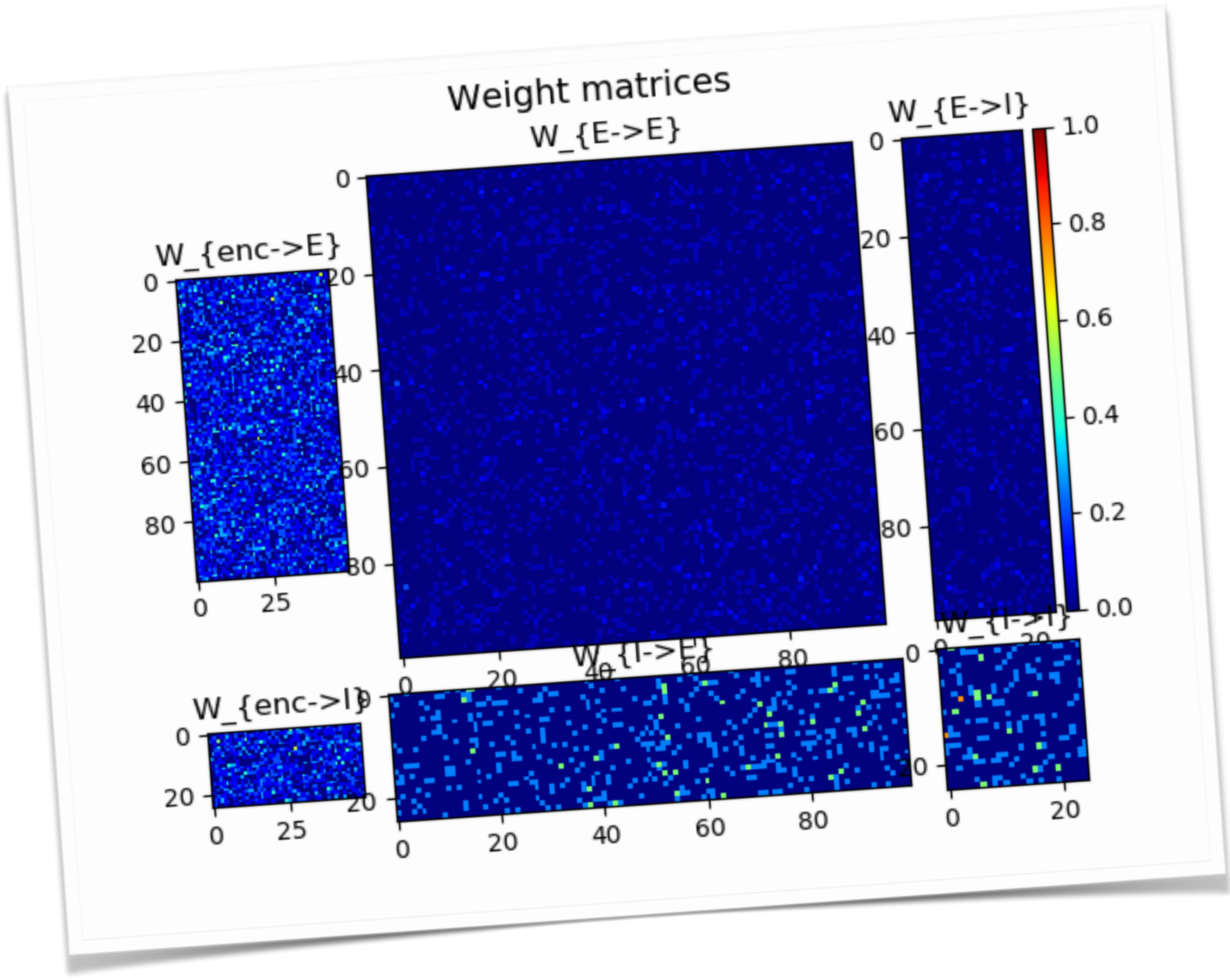
g (rel) inhibitory weights (-g * w)

RNN ex neurons

RNN in neurons

synaptic parameters

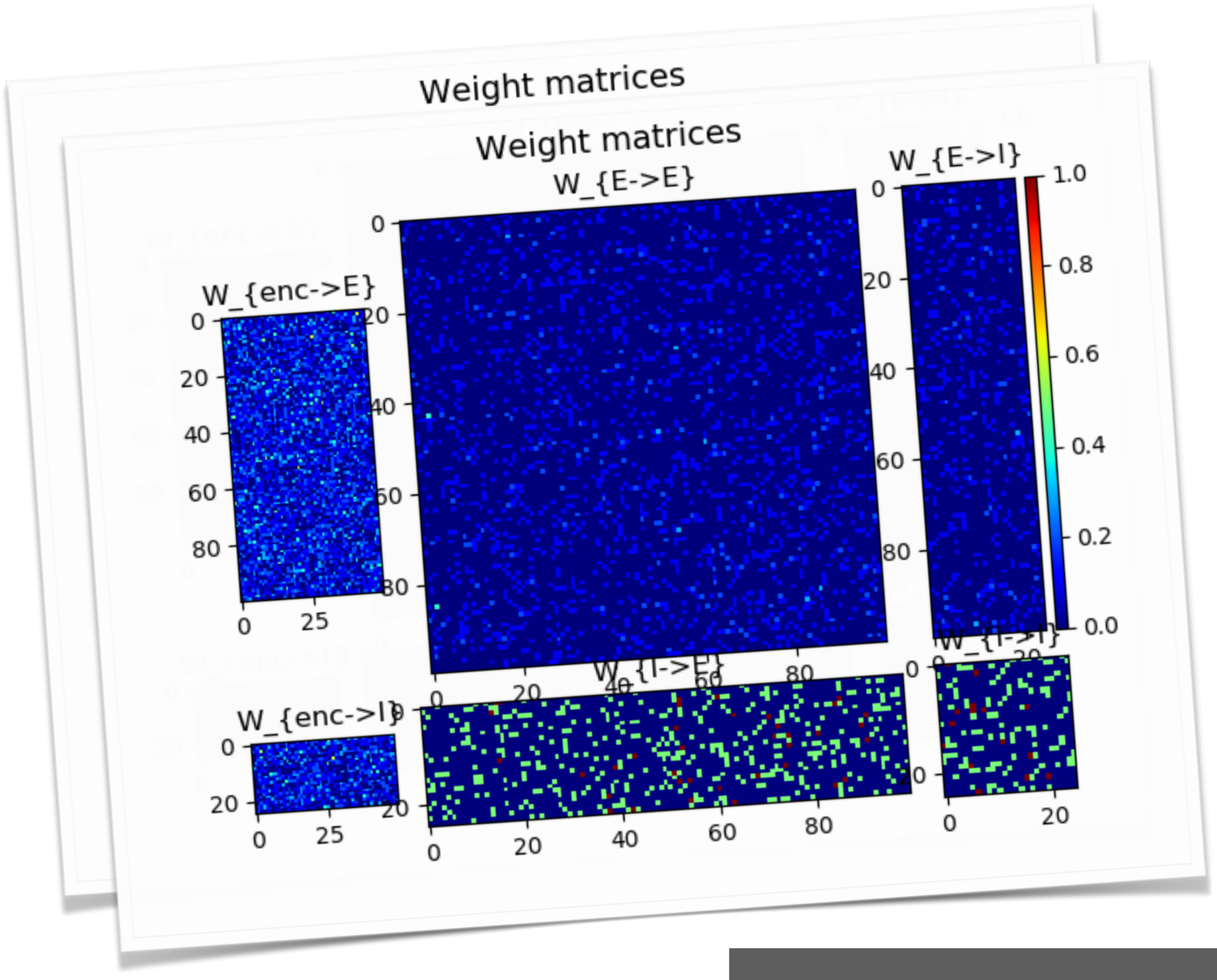
NS 10



w = 0.05

w parameter

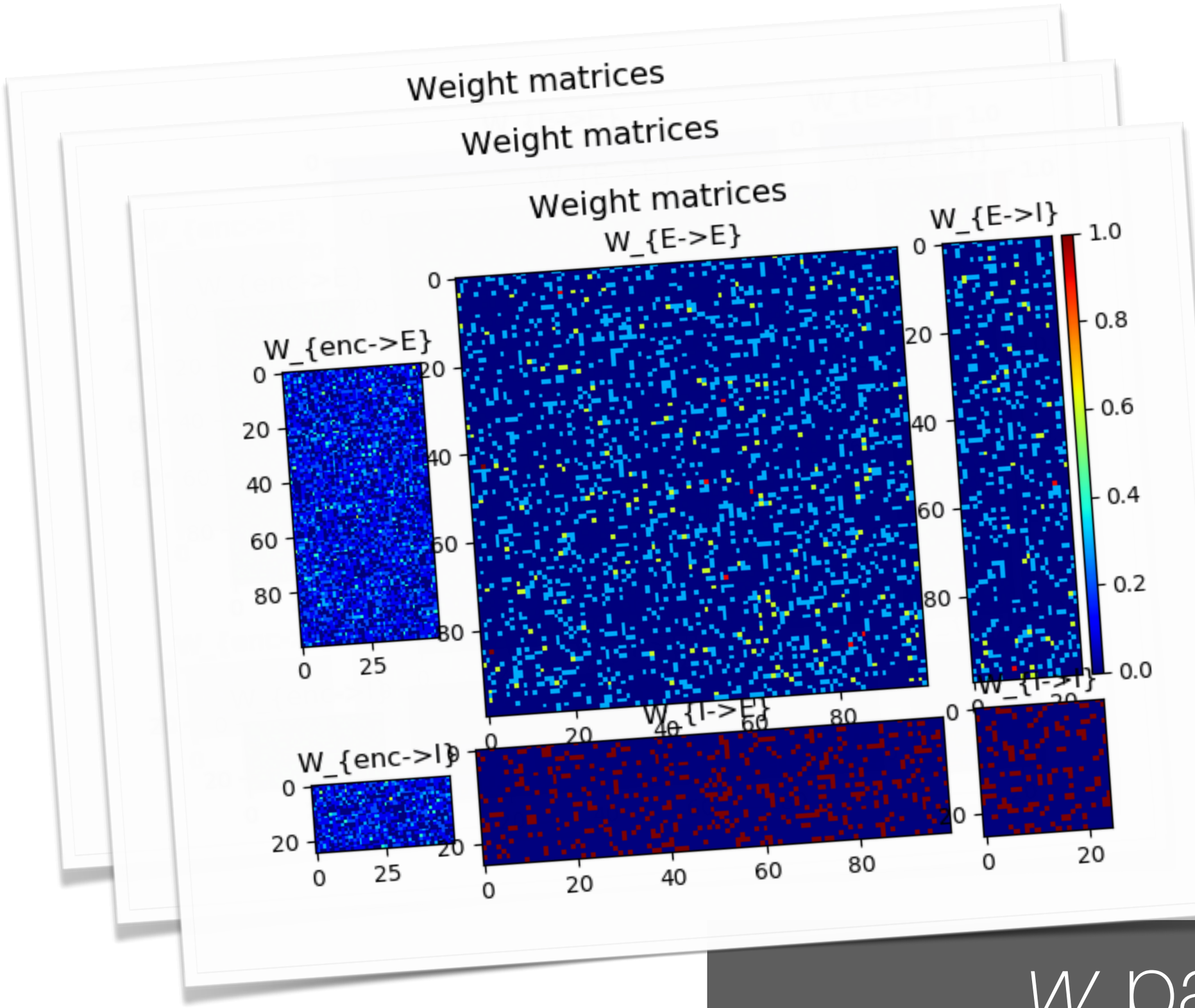
NS 10



w = 0.1

w parameter

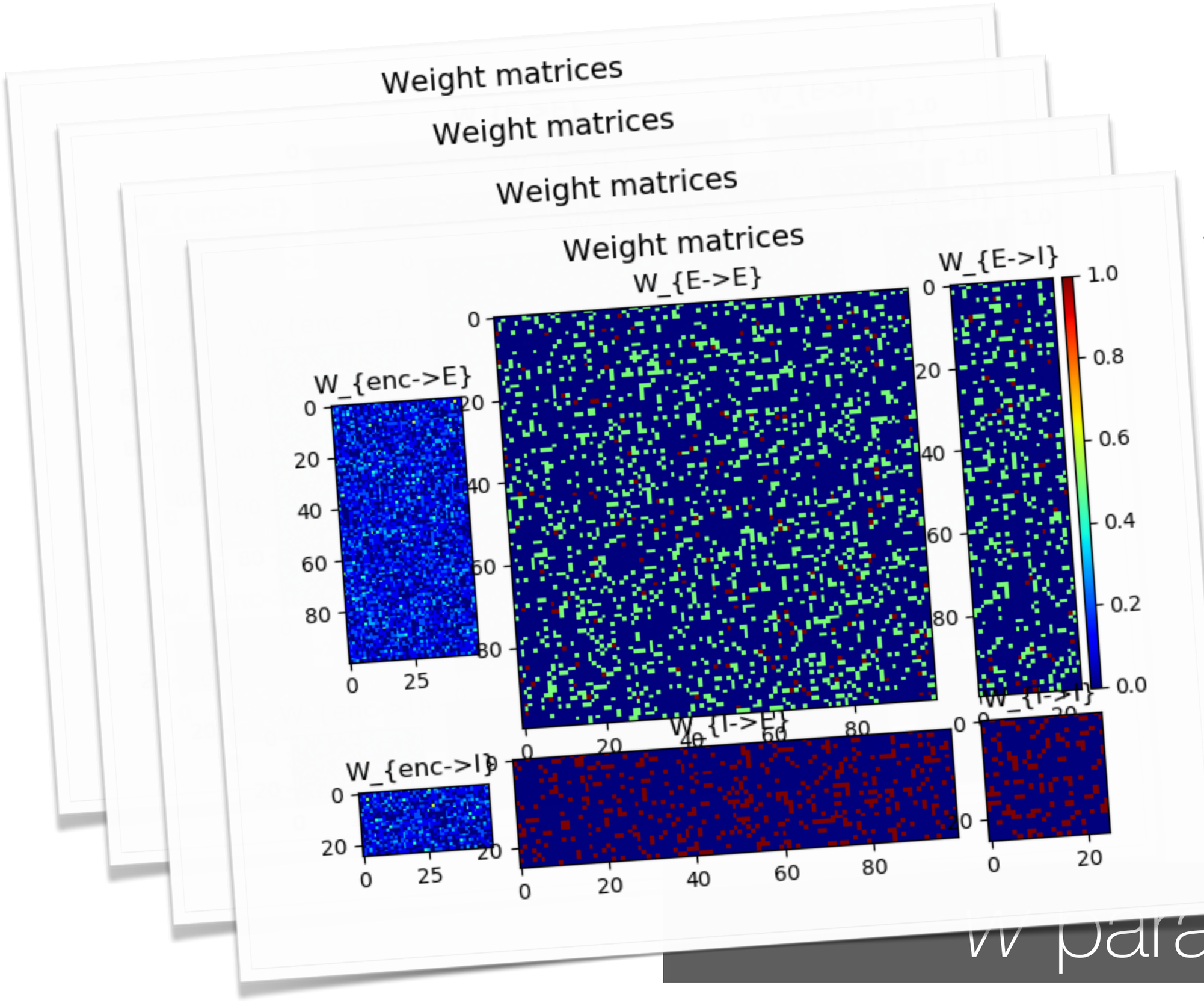
NS 10



w = 0.3

w parameter

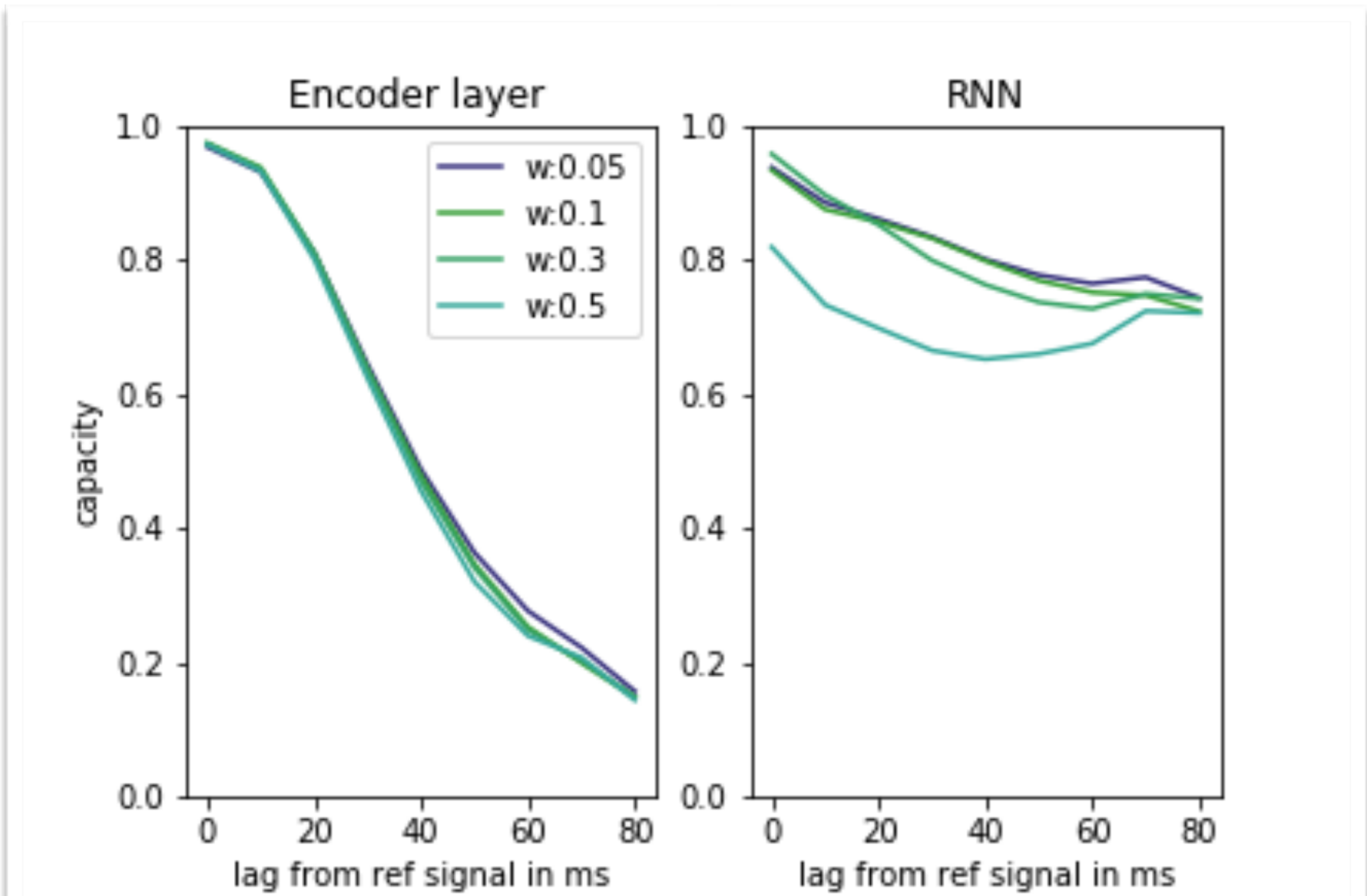
NS 10



w = 0.5

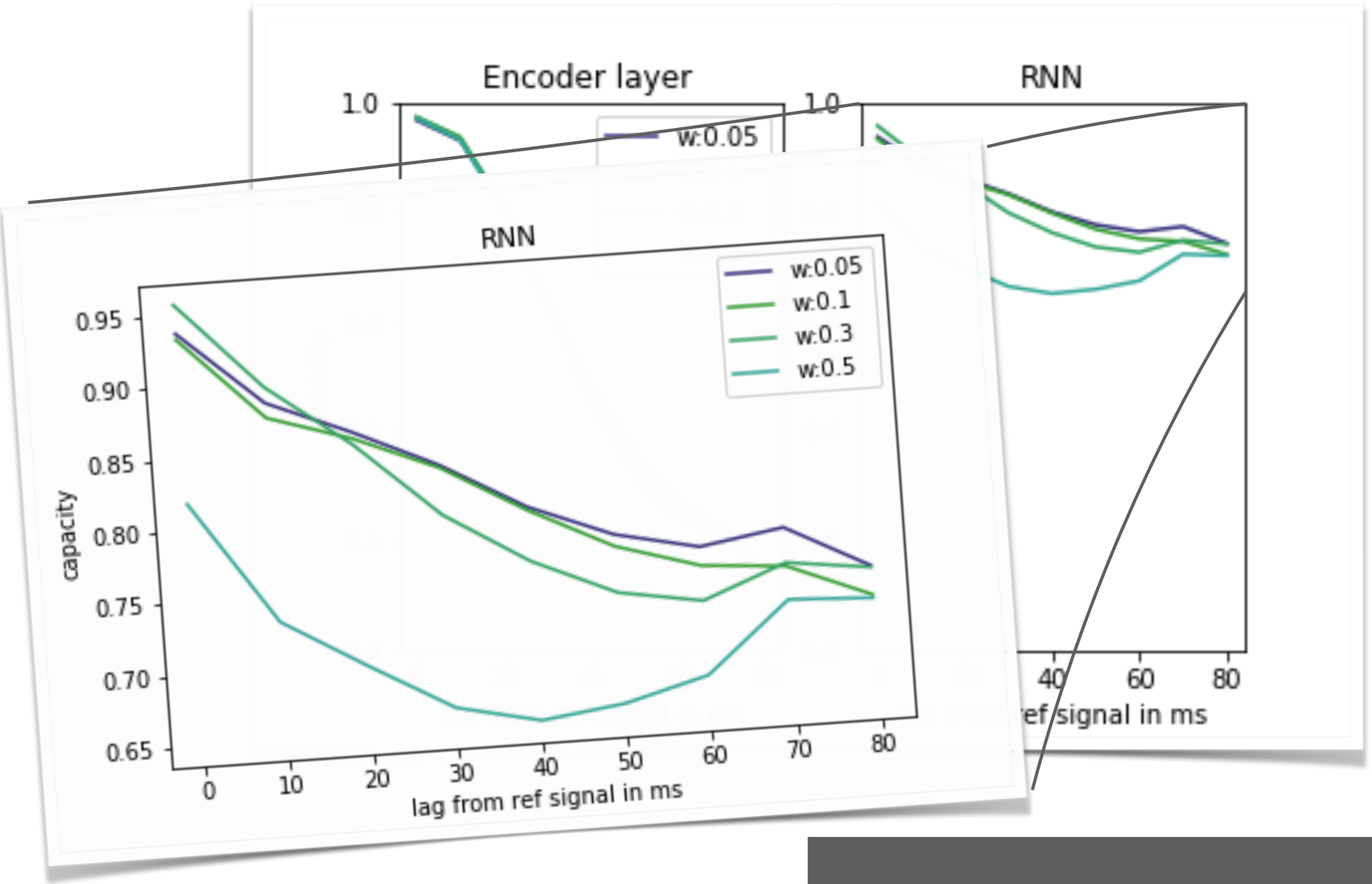
W parameter

NS 10

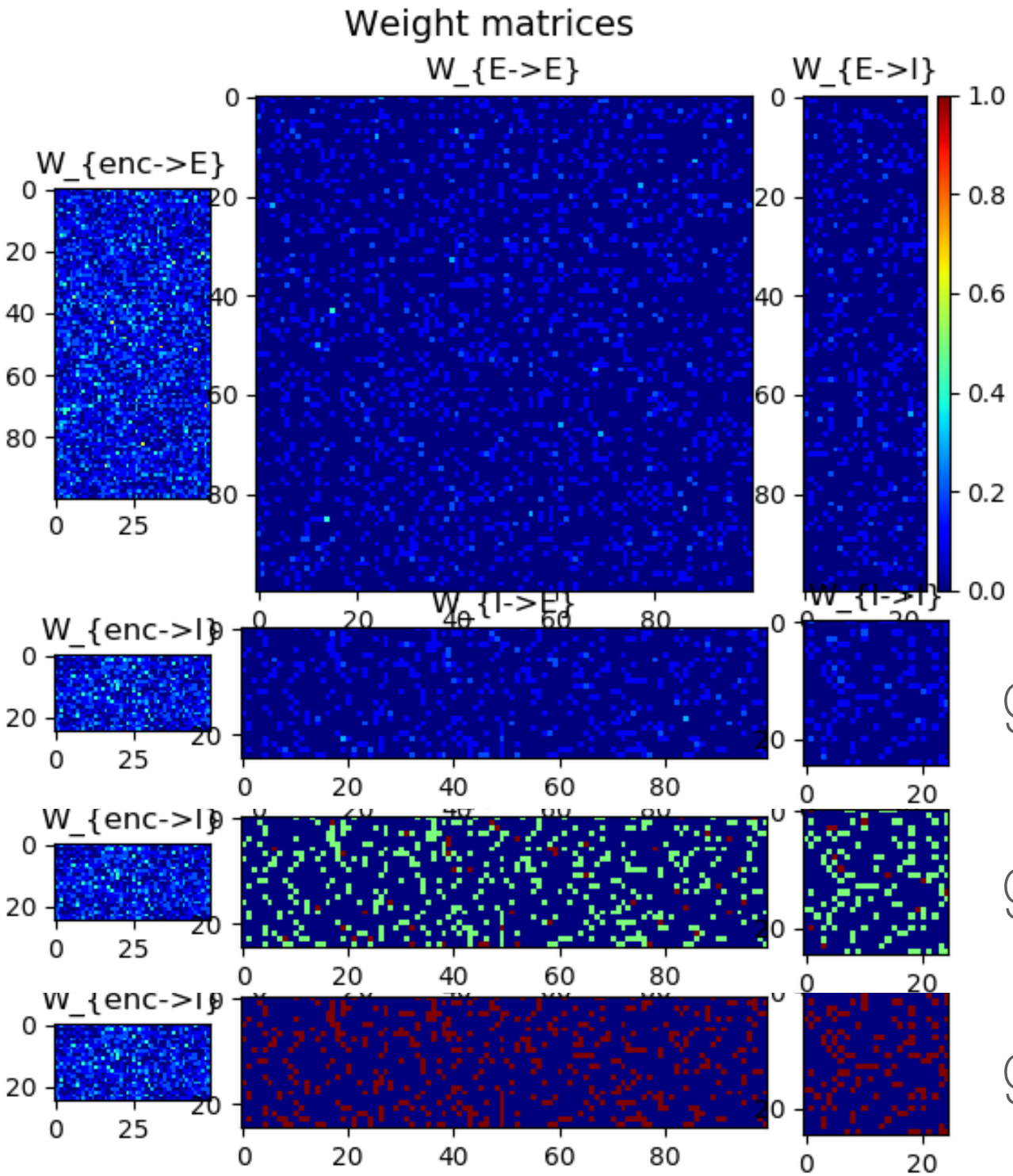


w results

NS 10



w results



g = 1

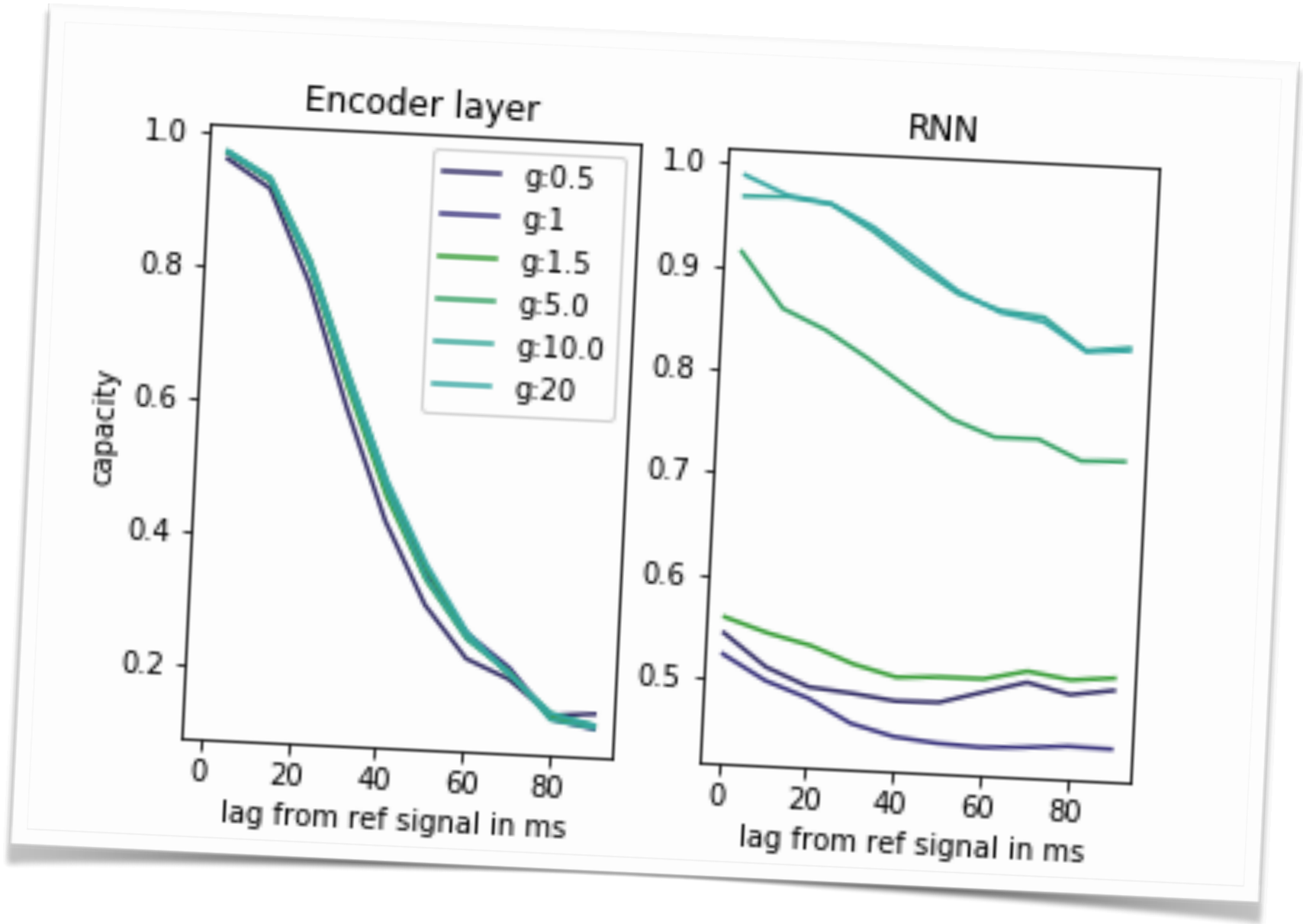
g = 5

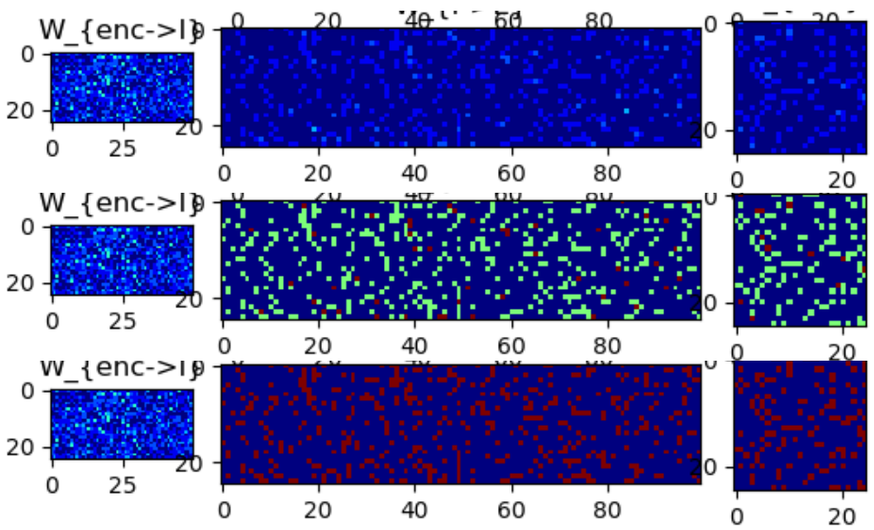
g = 10

g parameter

NS 10

NS 10



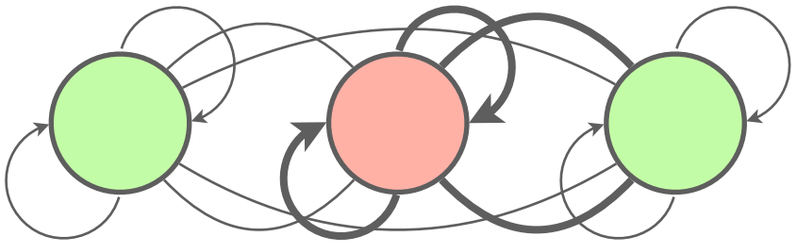


g = 1

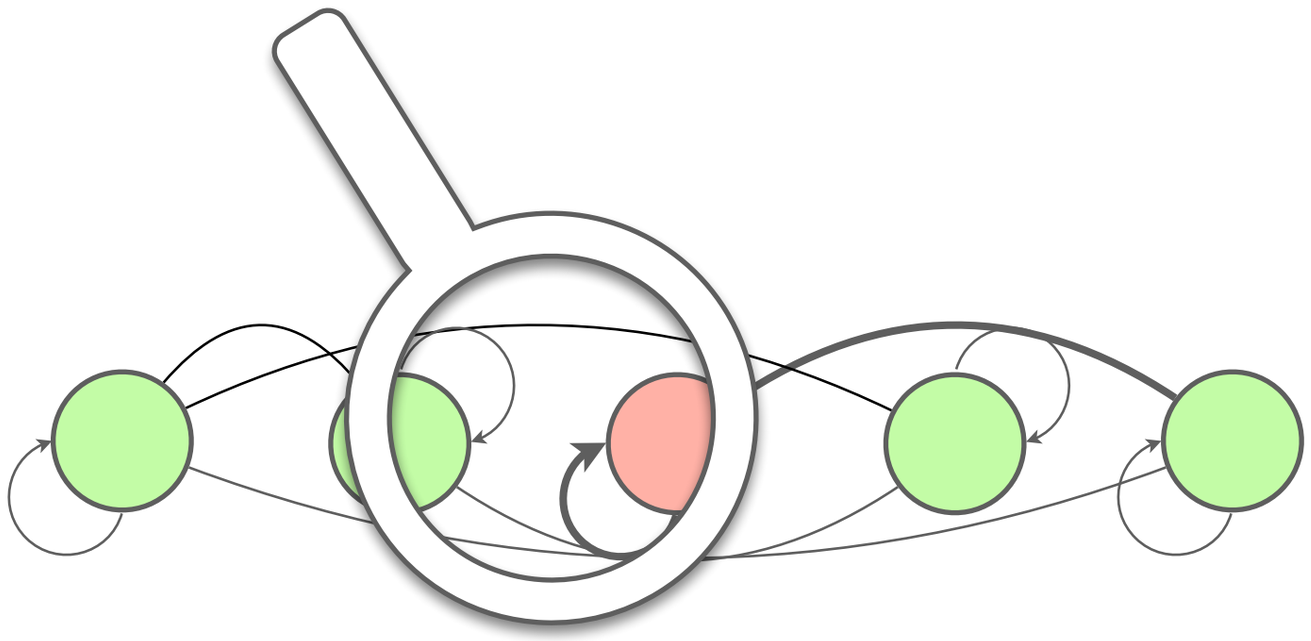
g = 5

g = 10

g results



in-degree

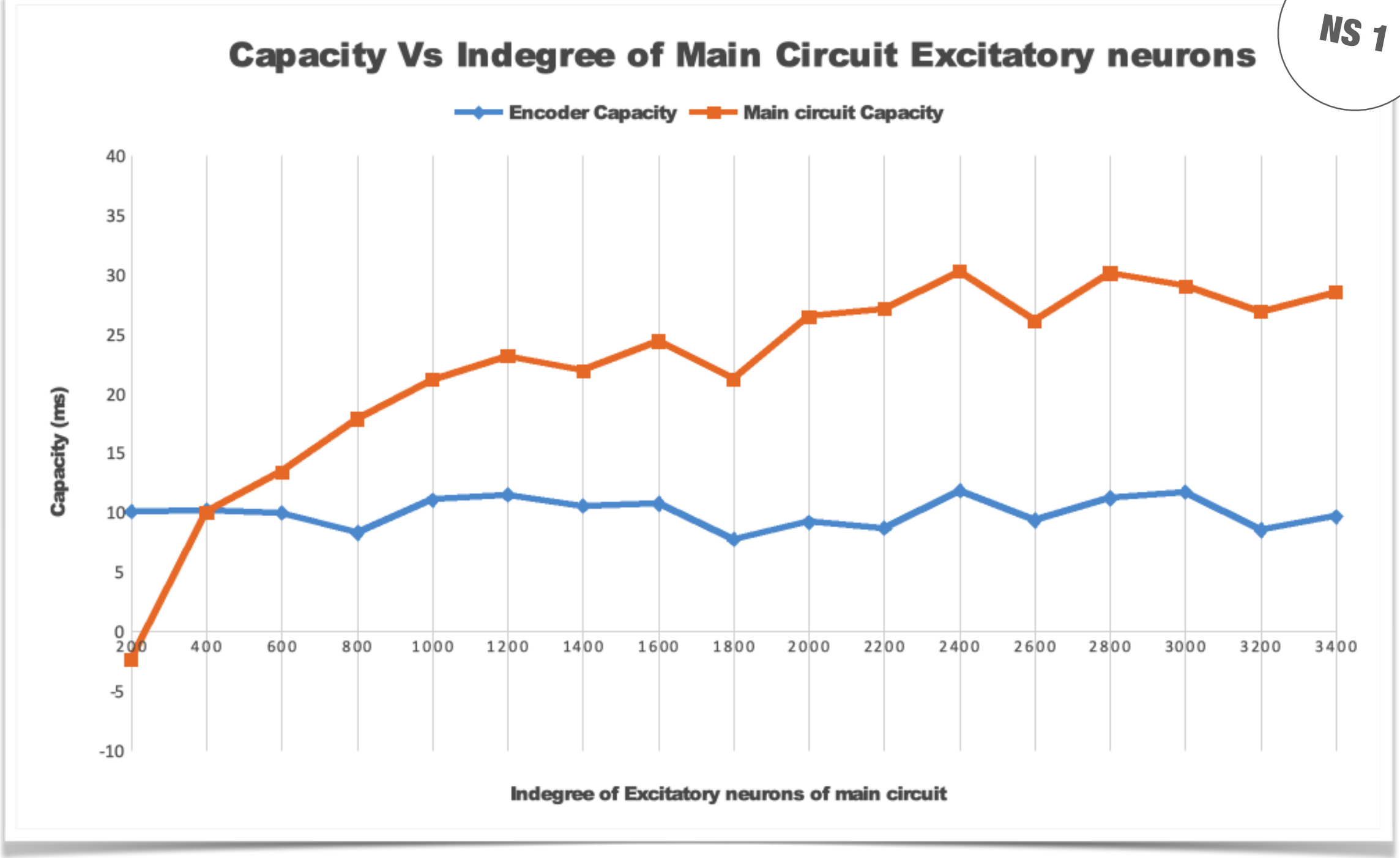


RNN ex neurons

RNN in neurons

connectivity

NS 1

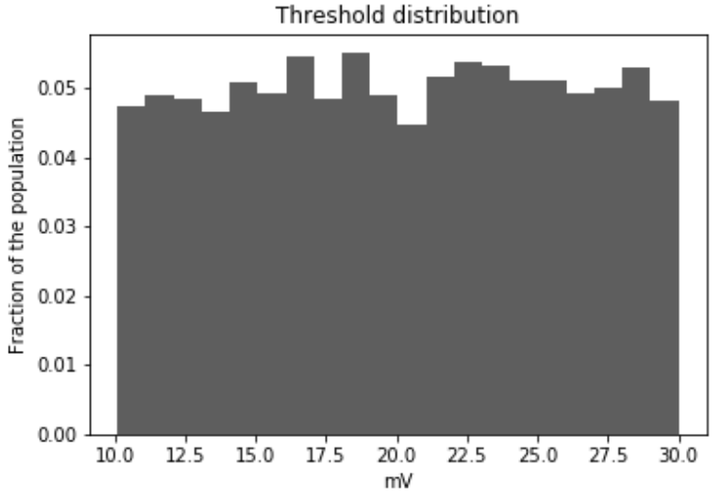

Capacity Vs Indegree of Main Circuit Excitatory neurons
Encoder Capacity
Main circuit Capacity
Capacity (ms)
40
35
30
25
20
15
10
5
0
-5
-10
200
400
600
800
1000
1200
1400
1600
1800
2000
2200
2400
2600
2800
3000
3200
3400
Indegree of Excitatory neurons of main circuit

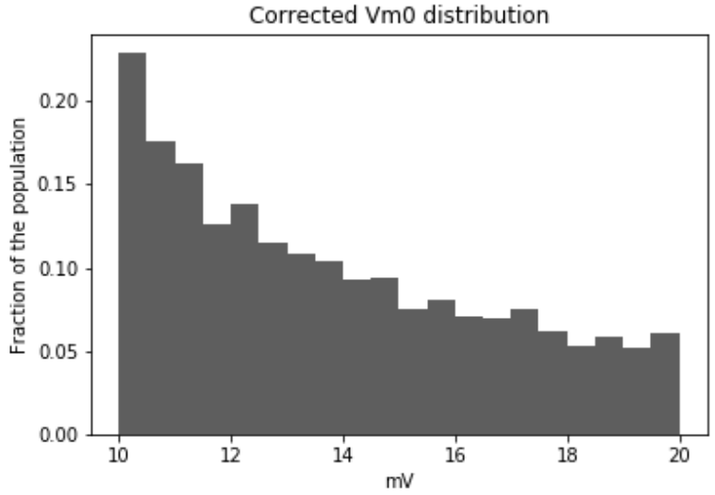
results

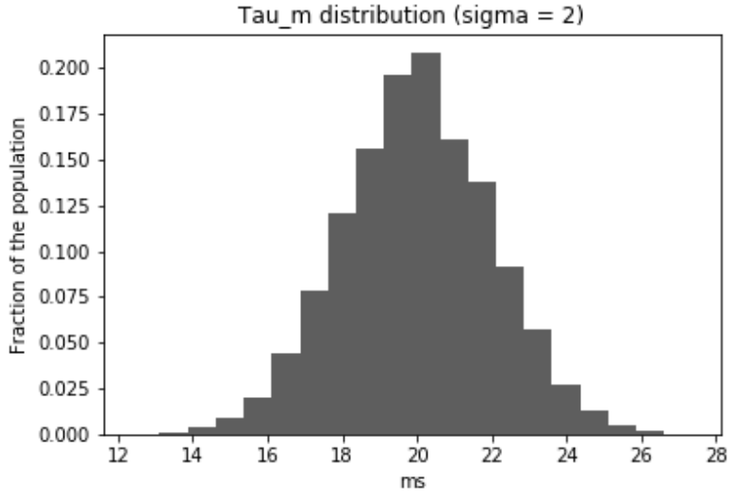
Th

Vm_0

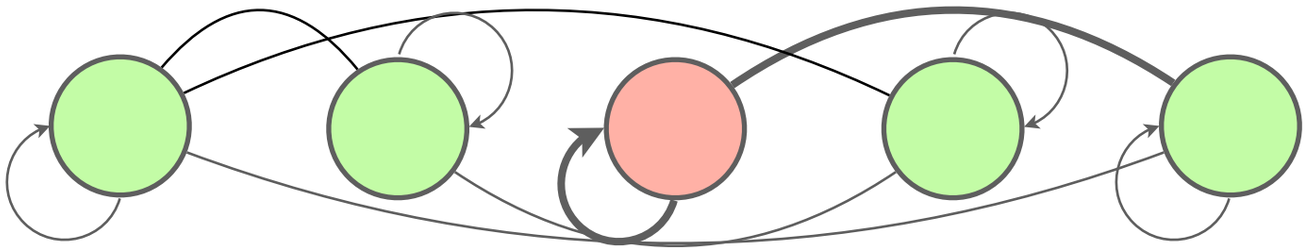
τ_m







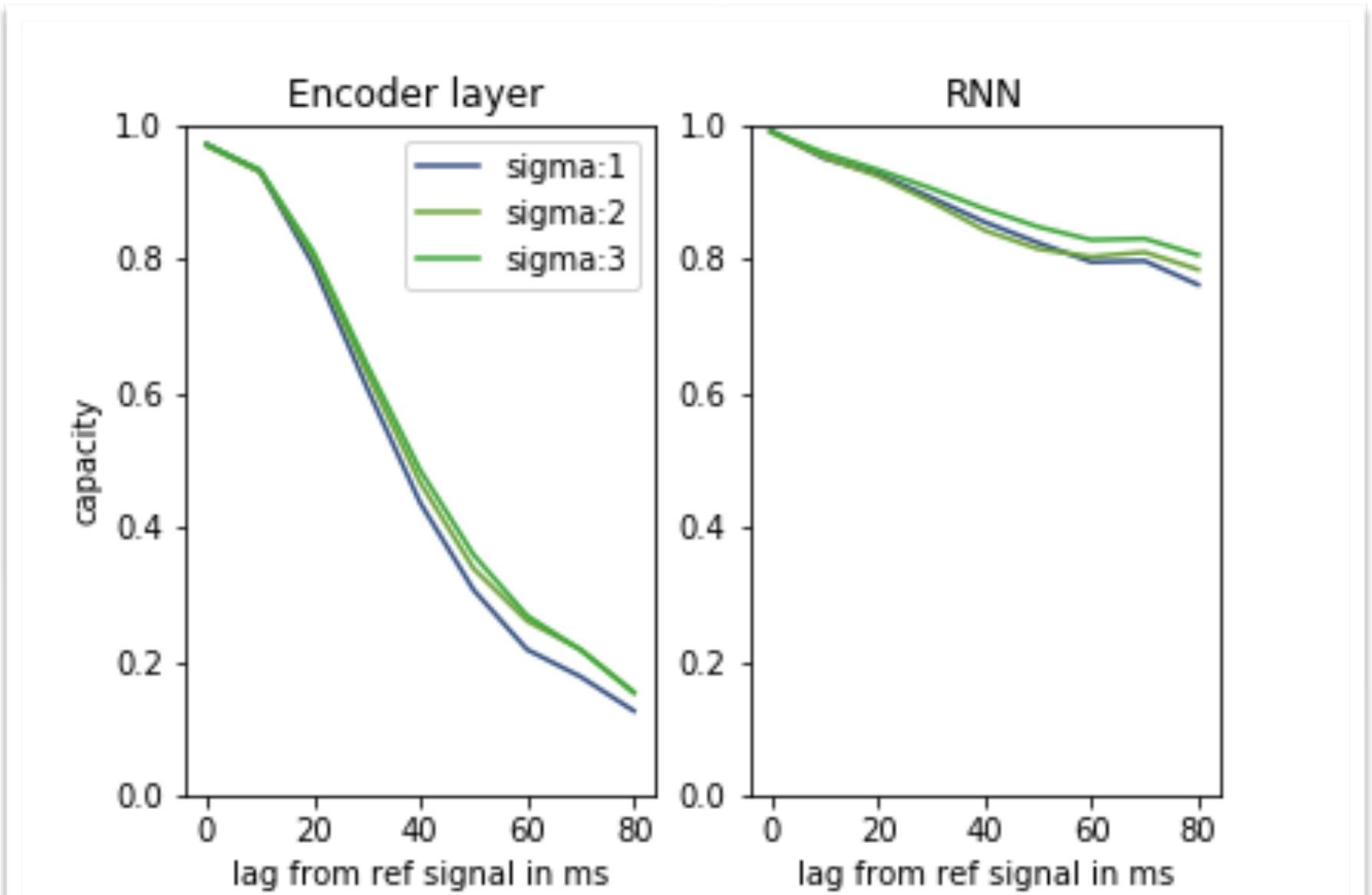
RNN ex neurons



RNN in neurons

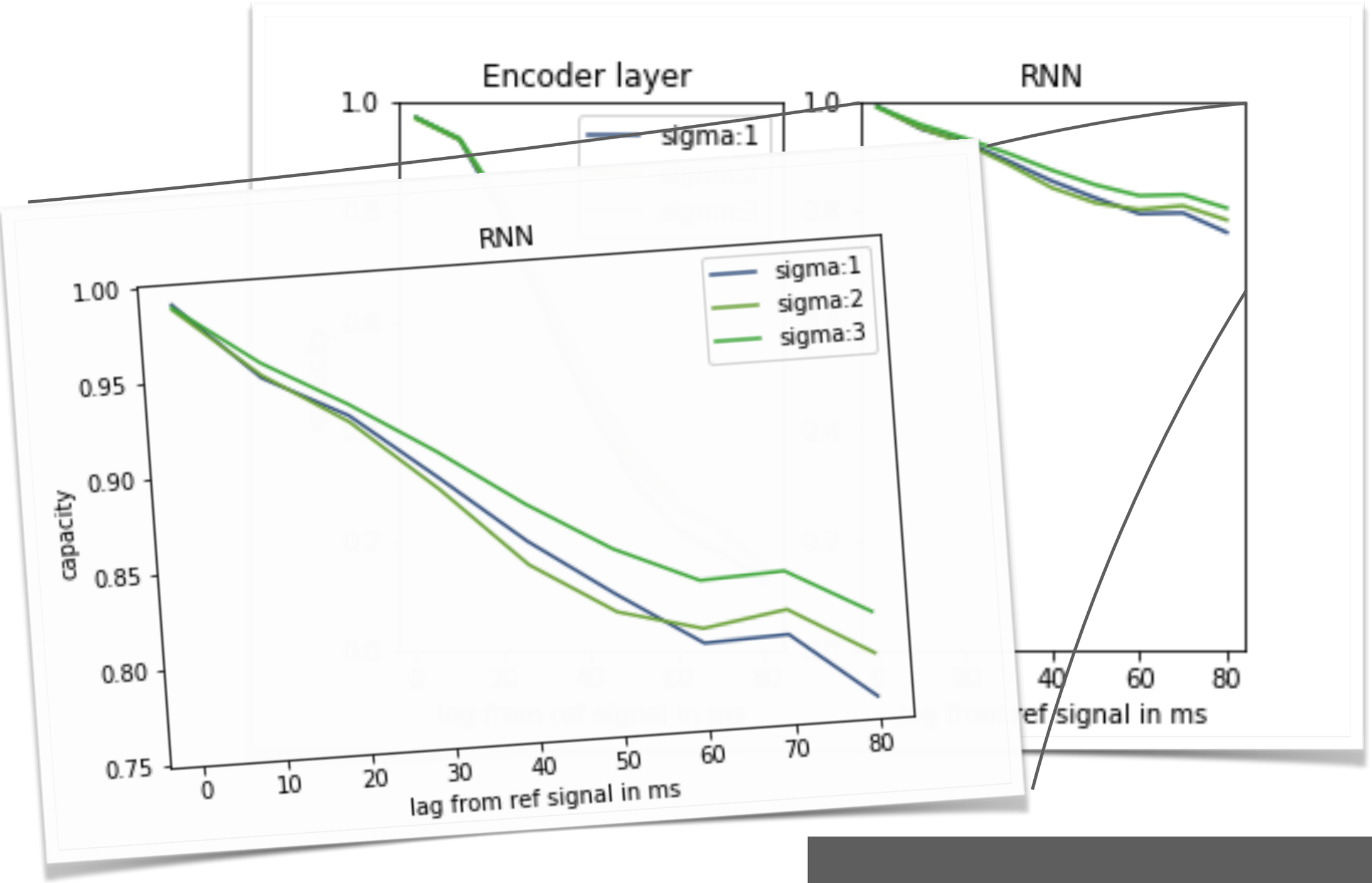
heterogeneity

NS 10

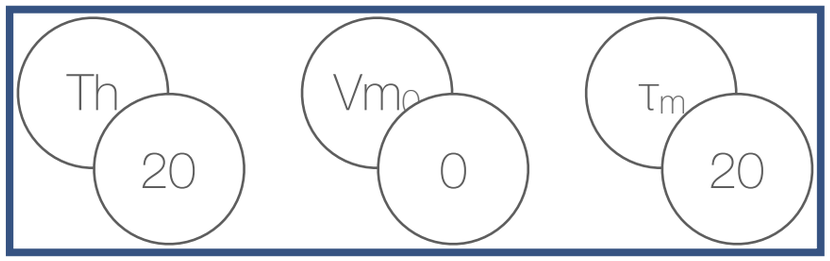


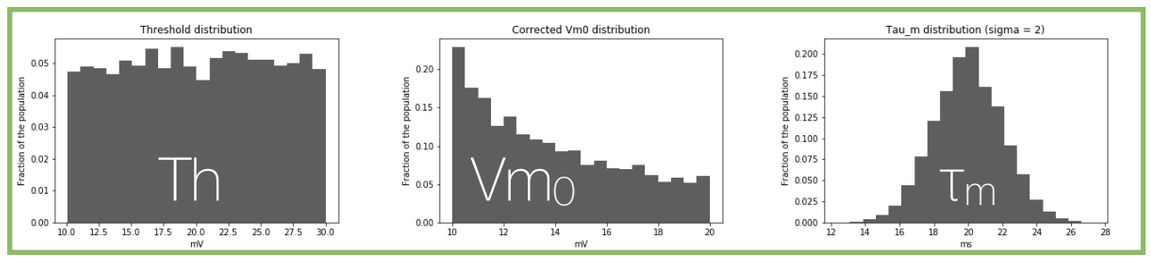
results

NS 10

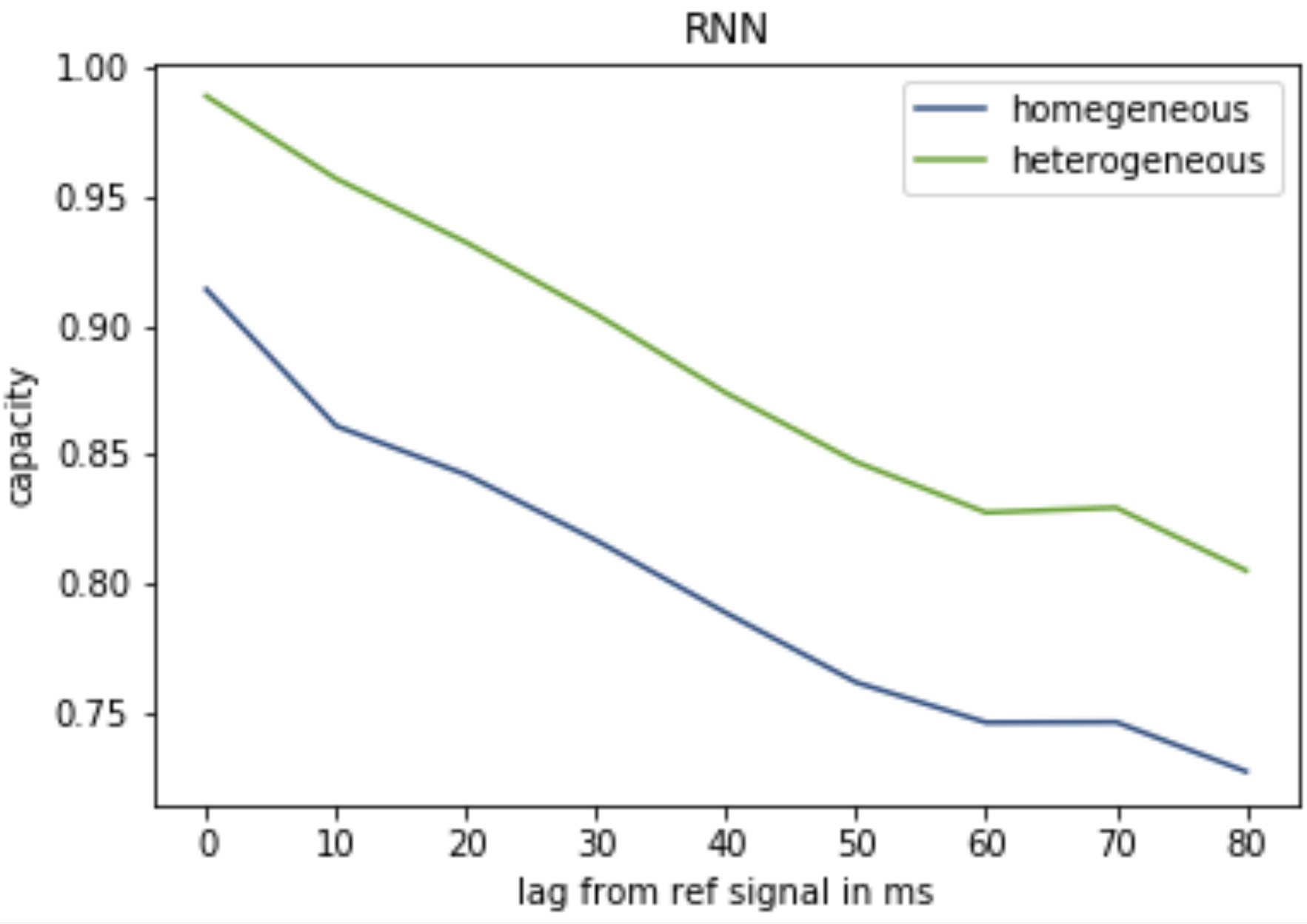


results





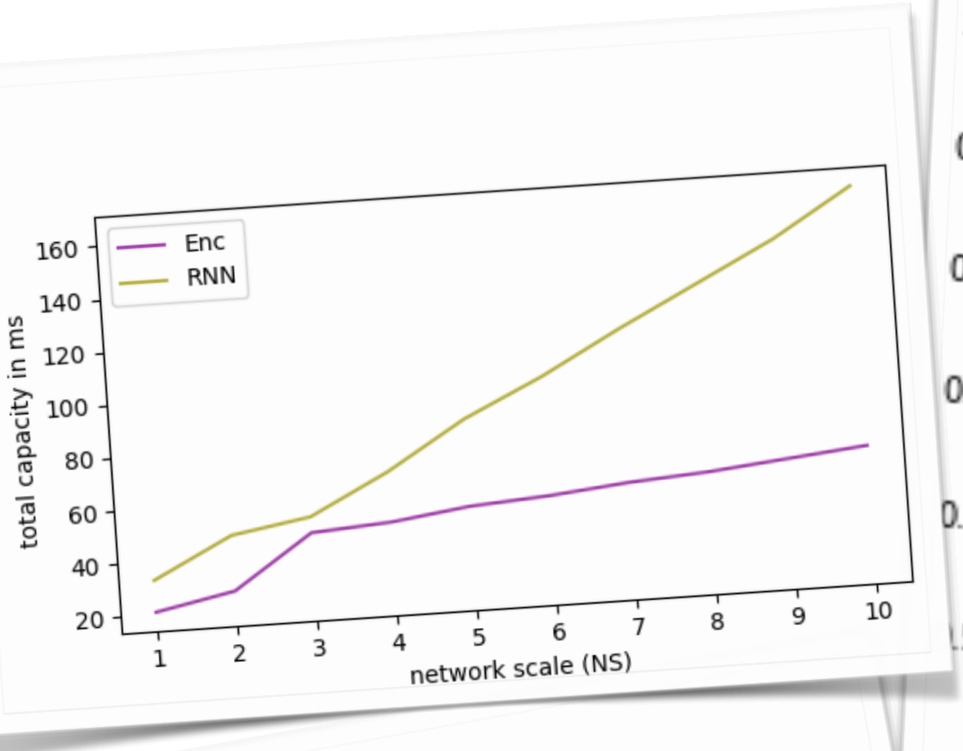
NS 10

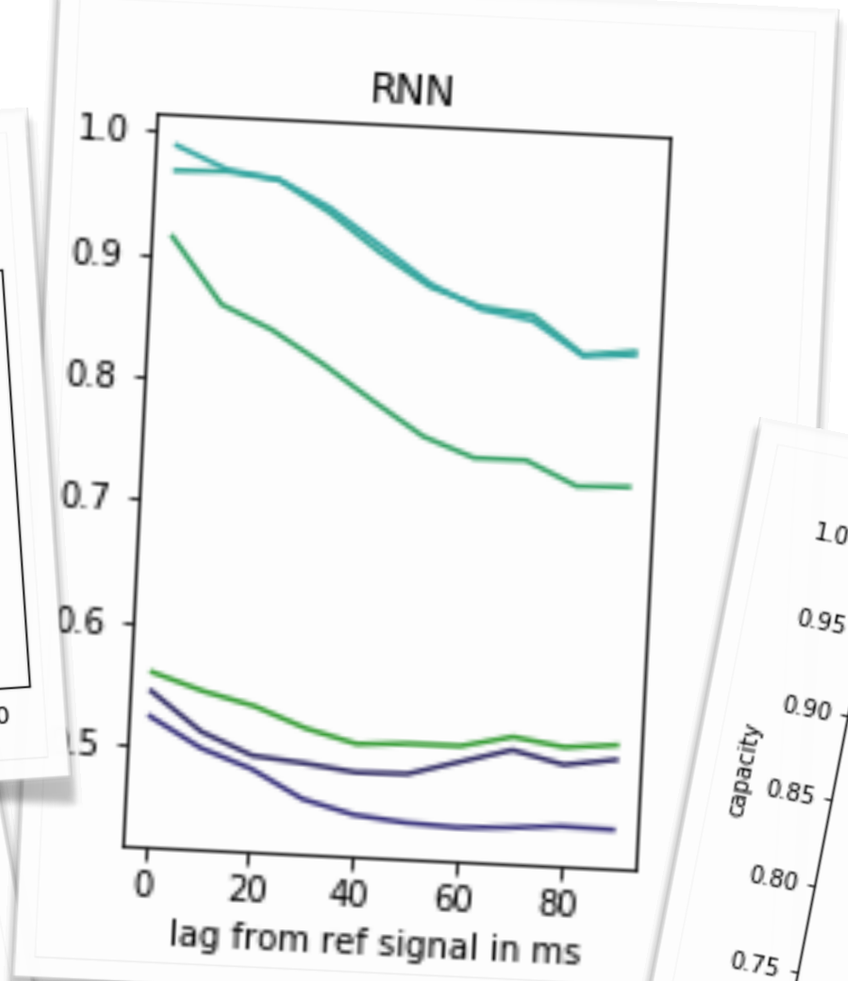


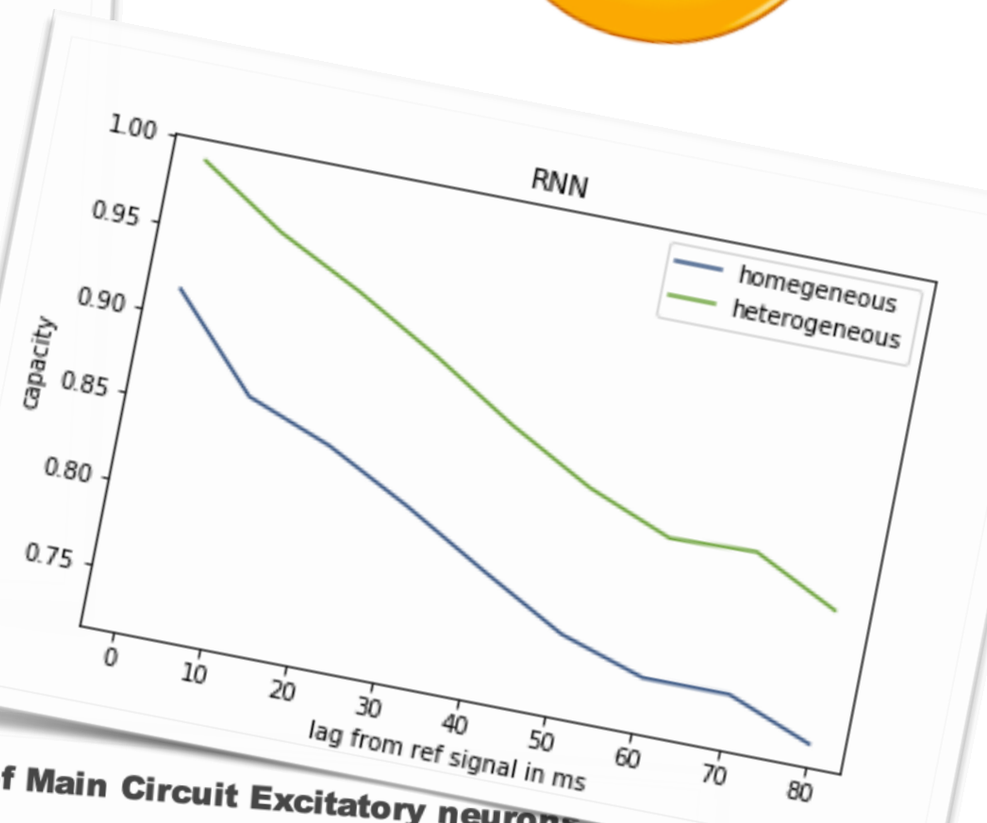
results

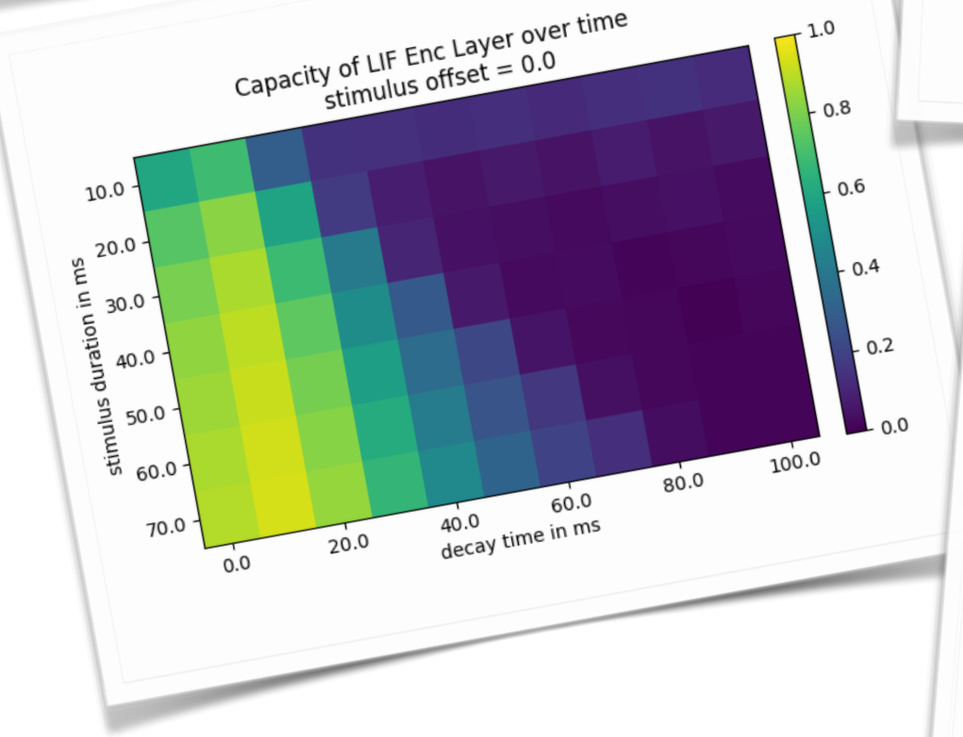
Roman's part

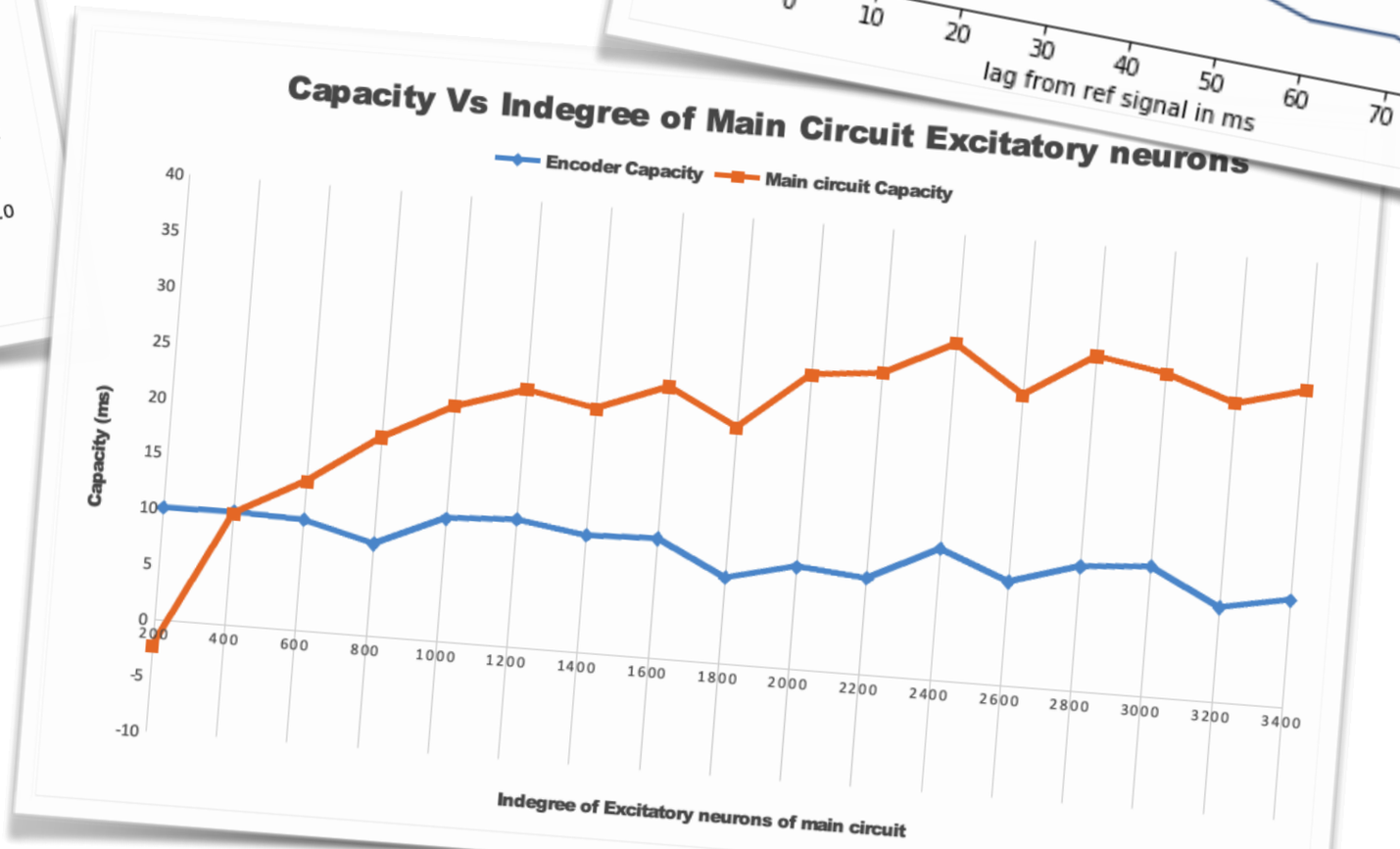
conclusions & remarks

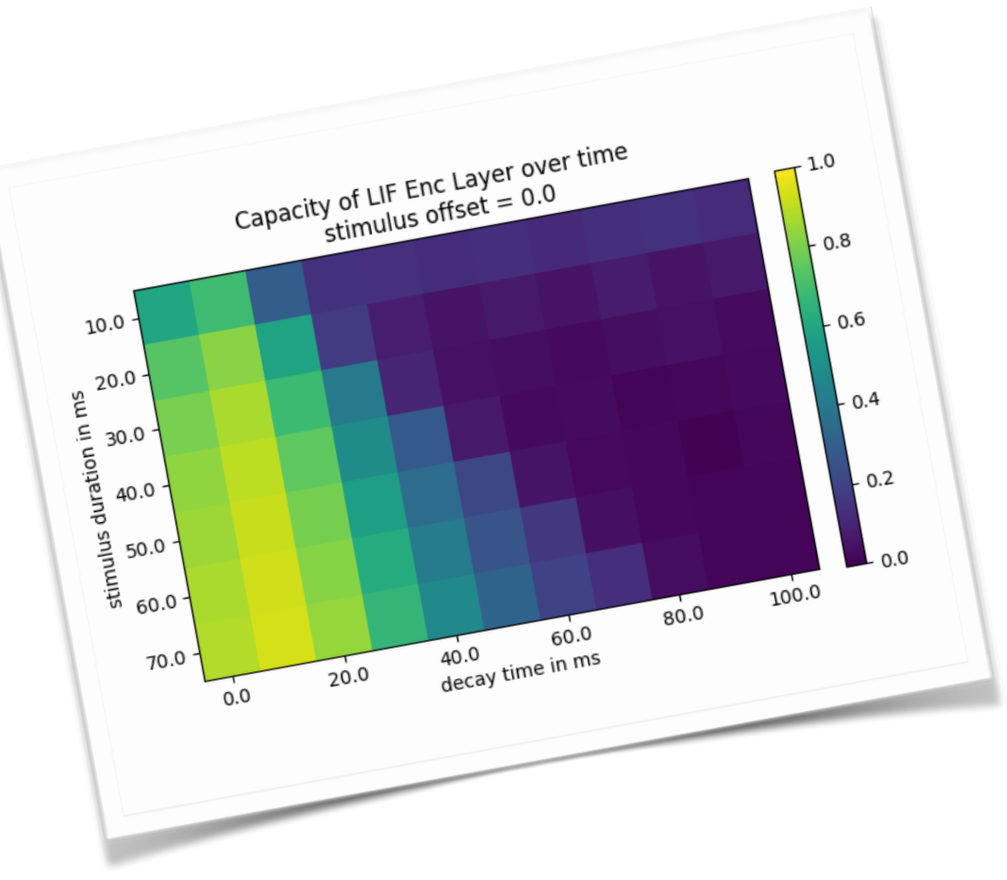

Enc
RNN
total capacity in ms
network scale (NS)

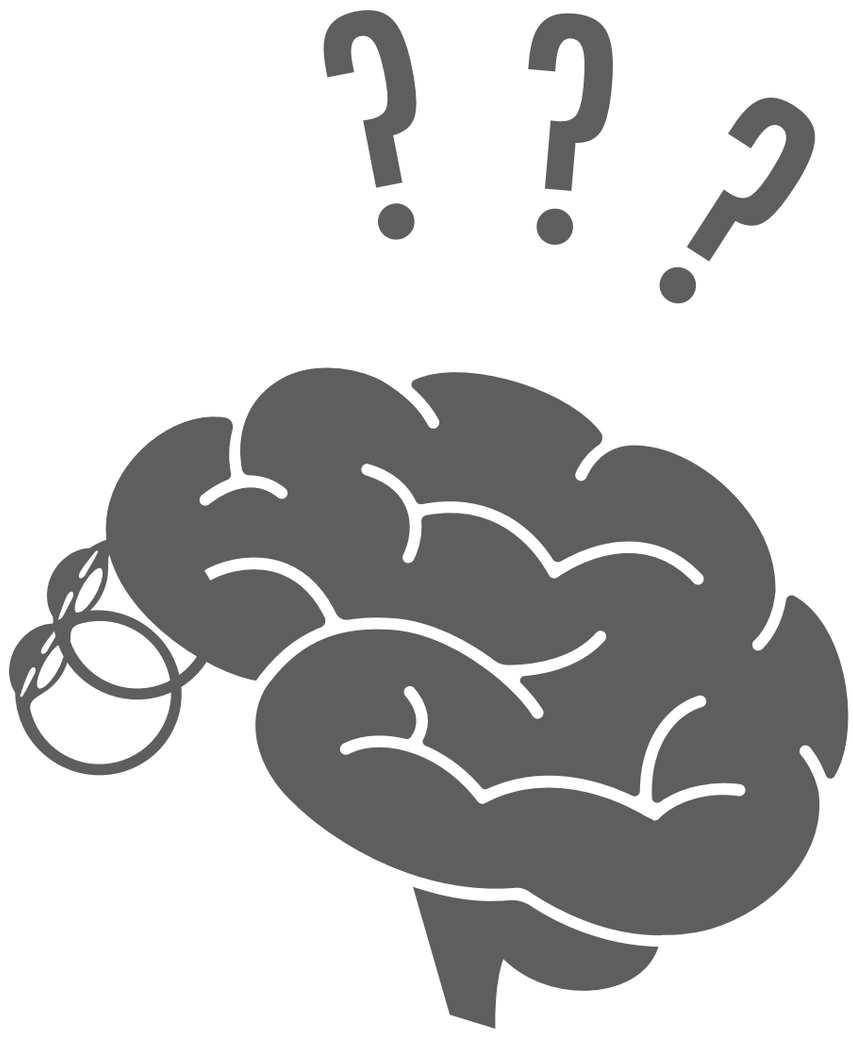

RNN
lag from ref signal in ms


RNN
homegeneous
heterogeneous
capacity
lag from ref signal in ms

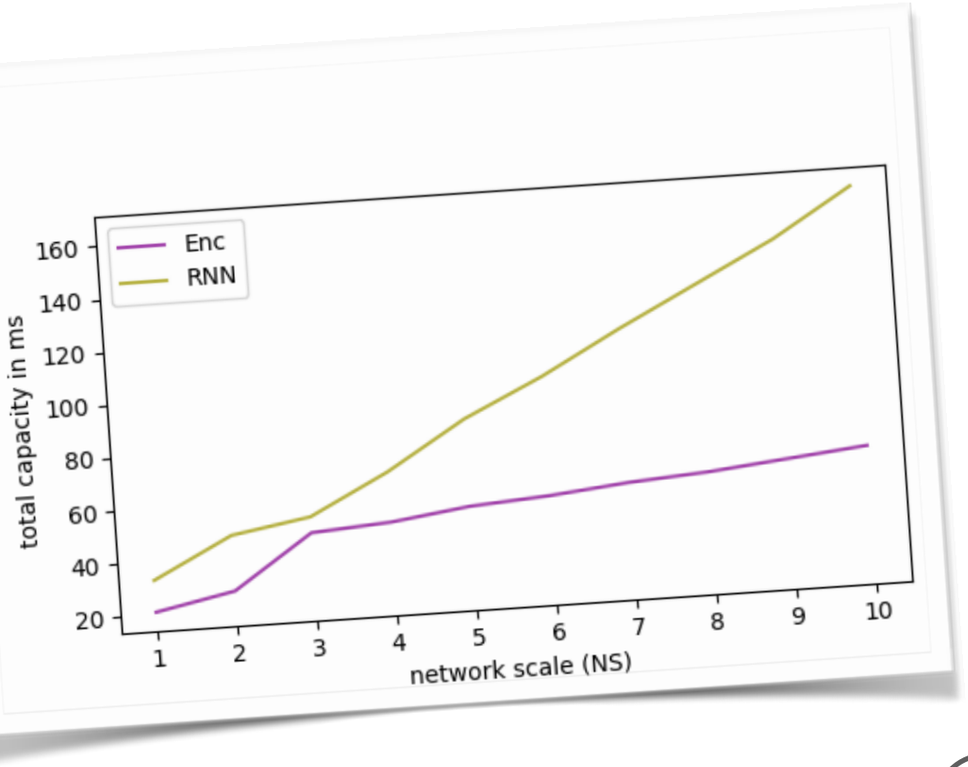

Capacity of LIF Enc Layer over time
stimulus offset = 0.0
stimulus duration in ms
decay time in ms


Capacity Vs Indegree of Main Circuit Excitatory neurons
Encoder Capacity
Main circuit Capacity
Capacity (ms)
Indegree of Excitatory neurons of main circuit

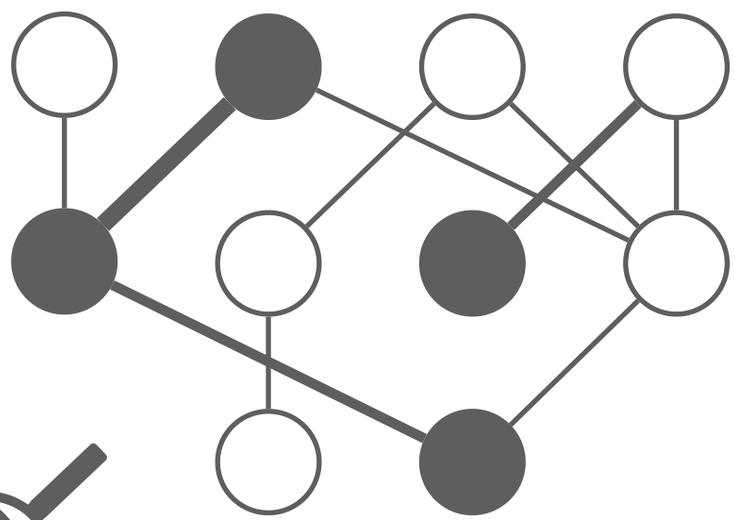


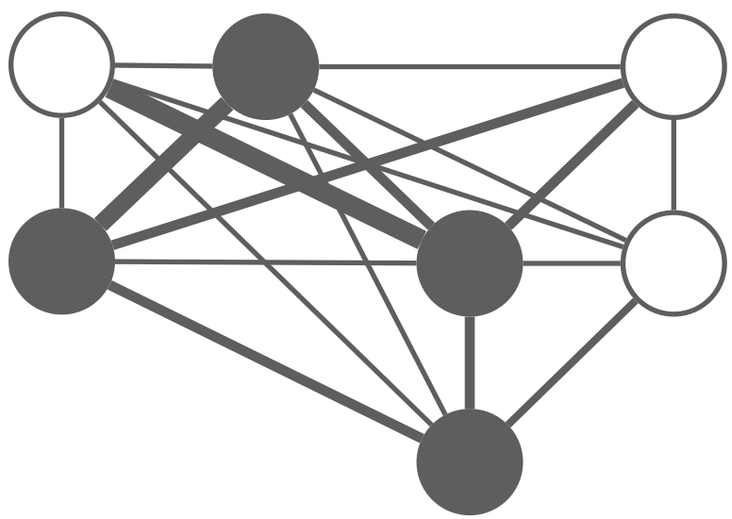


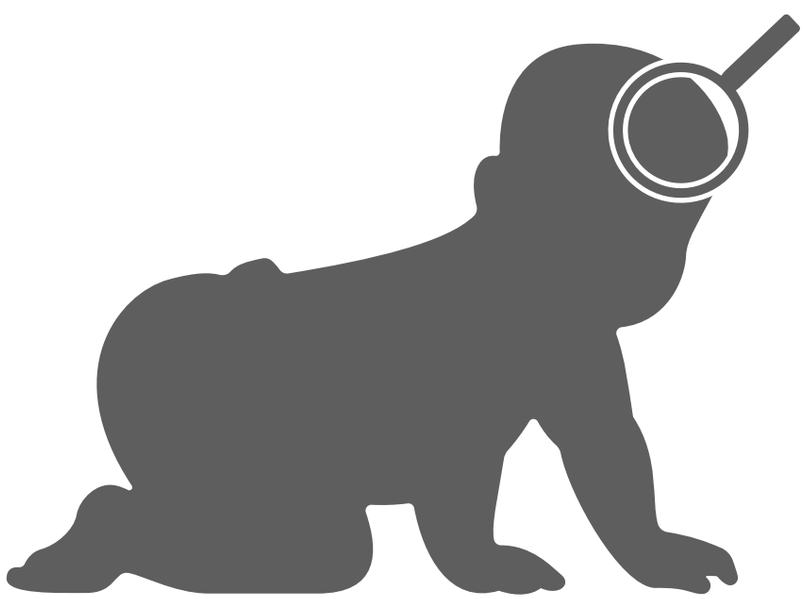
stimulus duration impacts capacity

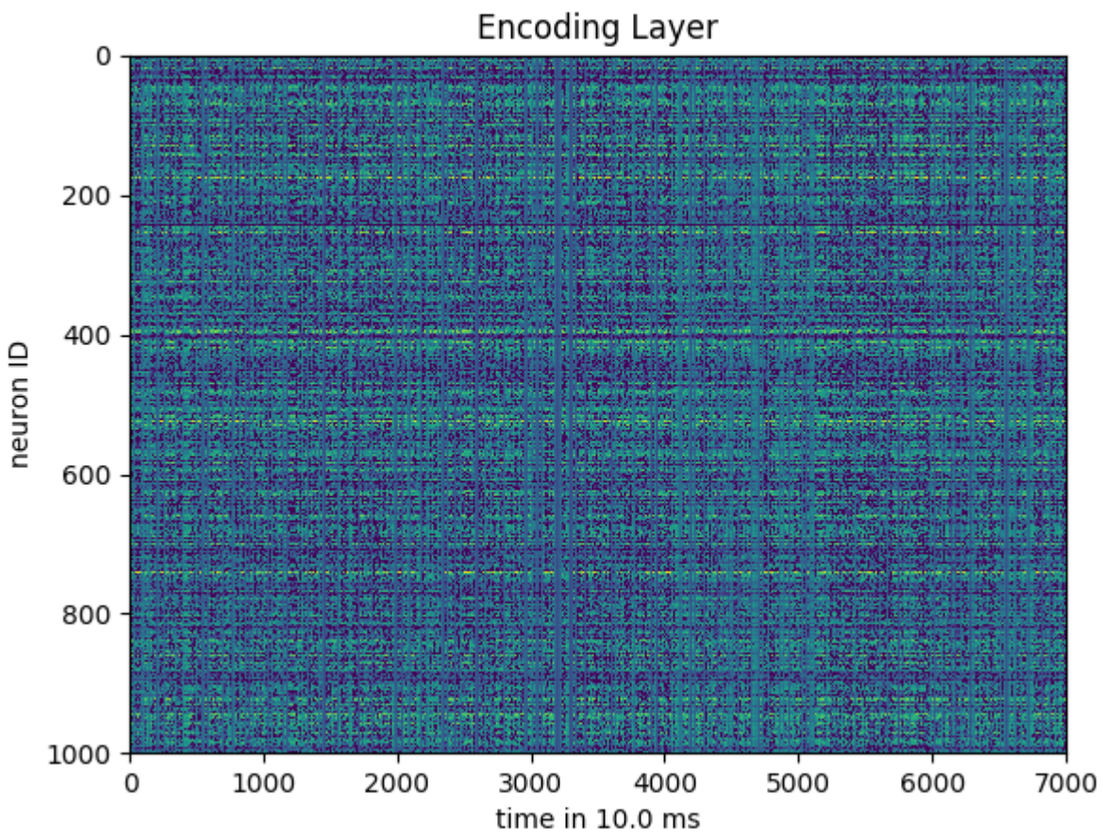


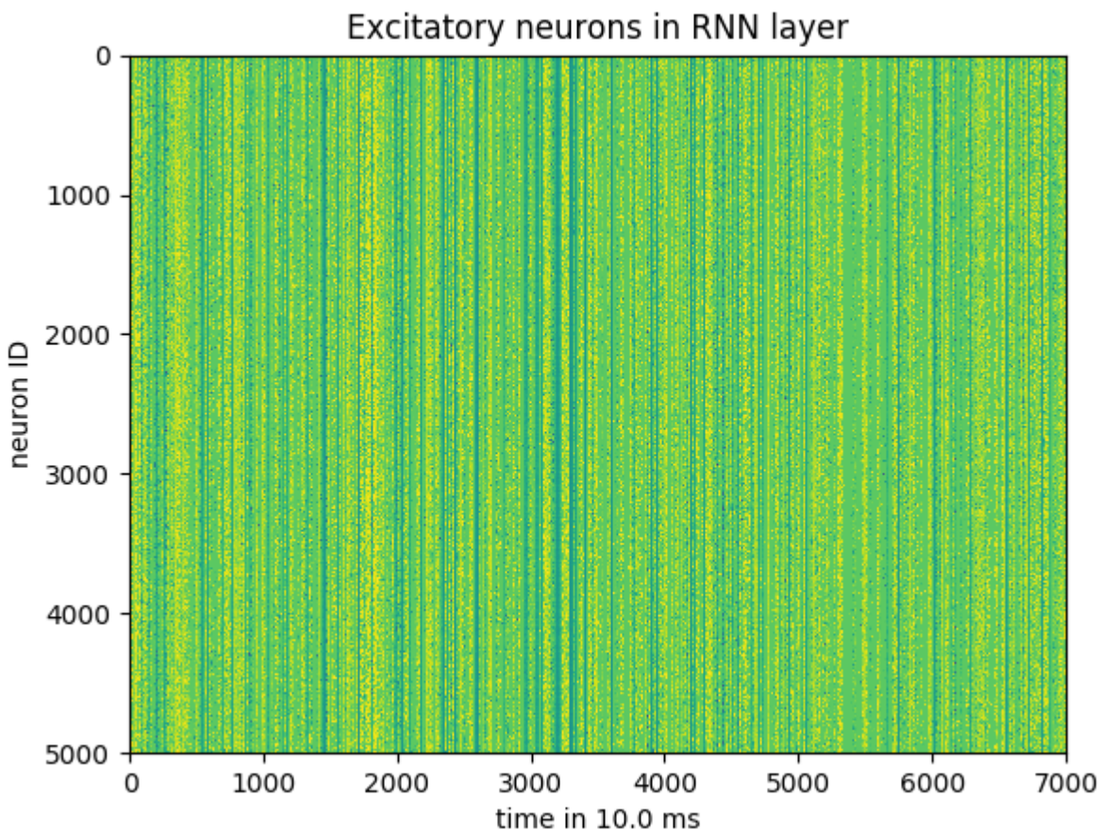
what about plasticity ?











limitations

no **HPC** no **fun**

Job ID	Username	Queue	Jobname	SessID	NDS	TSK	Memory	Time	S	Time
35473493.dccn-l029.dcc	tomcla	batch	STDIN	107267	--	--	64gb	18:00:00	R	00:00:15
35473494.dccn-l029.dcc	tomcla	batch	STDIN	89530	--	--	64gb	18:00:00	R	00:00:15
35473495.dccn-l029.dcc	tomcla	batch	STDIN	113053	--	--	64gb	18:00:00	R	00:00:15
35473496.dccn-l029.dcc	tomcla	batch	STDIN	47615	--	--	64gb	18:00:00	R	00:00:15
35473497.dccn-l029.dcc	tomcla	batch	STDIN	143278	--	--	64gb	18:00:00	R	00:00:15
35473498.dccn-l029.dcc	tomcla	batch	STDIN	28376	--	--	64gb	18:00:00	R	00:00:15
35473499.dccn-l029.dcc	tomcla	batch	STDIN	36917	--	--	64gb	18:00:00	R	00:00:14
35473500.dccn-l029.dcc	tomcla	batch	STDIN	36938	--	--	64gb	18:00:00	R	00:00:14
35473501.dccn-l029.dcc	tomcla	batch	STDIN	36979	--	--	64gb	18:00:00	R	00:00:14
35473502.dccn-l029.dcc	tomcla	batch	STDIN	27737	--	--	64gb	18:00:00	R	00:00:14

limitations

Renato Duarte

Alexander van Meegen