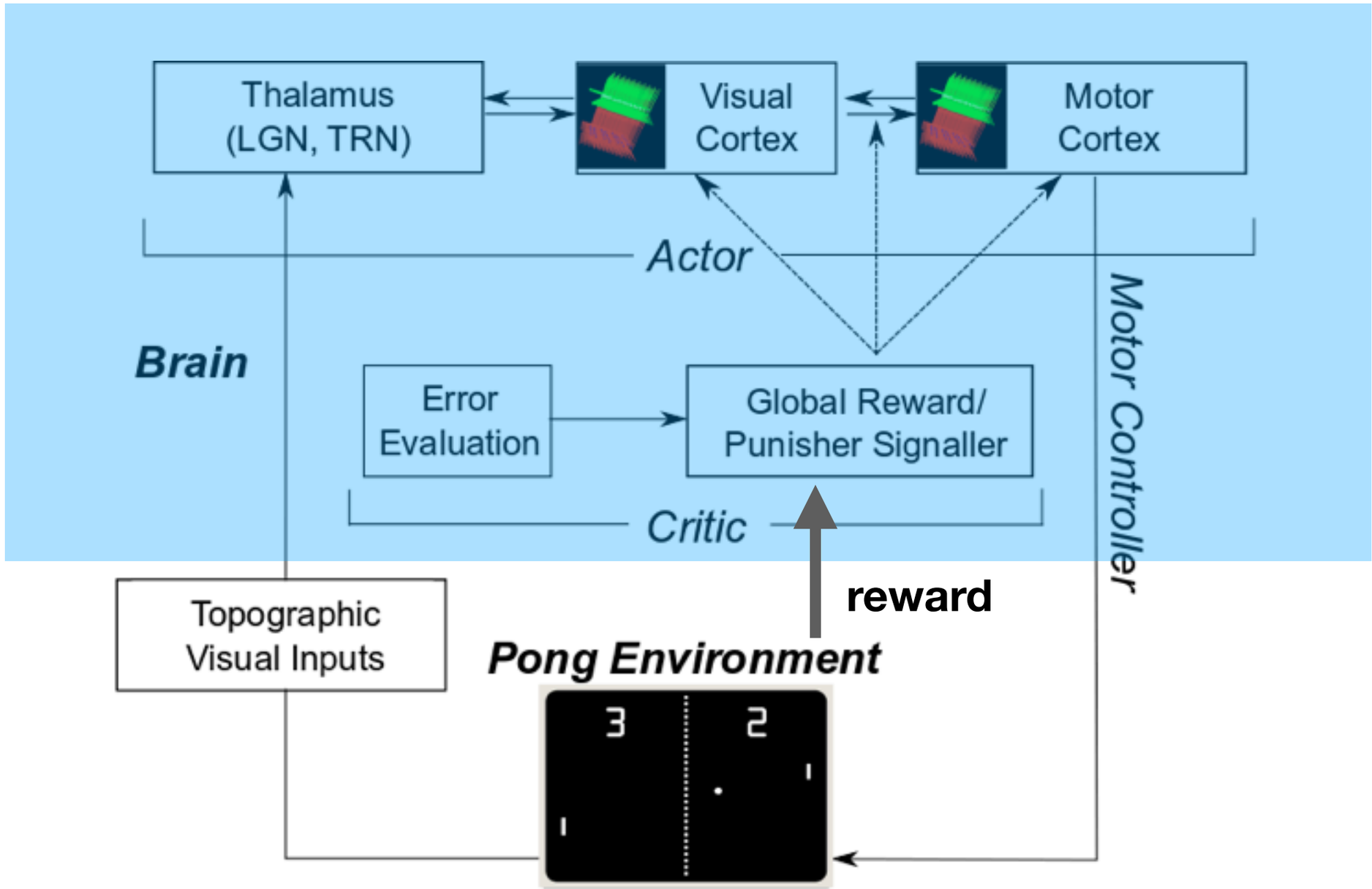
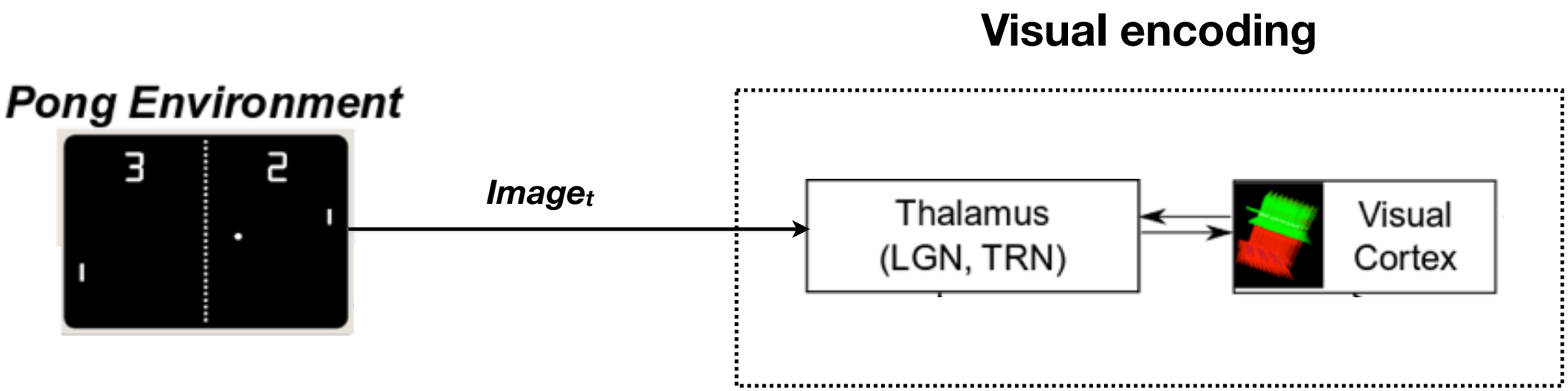
Schematic of closed-loop brain model training



T1: The actor/agent needs to interpret the environment of the game

Modeling visual pathway

- to have a representation of visual objects/pixels in the environment (unsupervised learning)
- to learn the changes in the environment e.g. direction of ball, direction of racket
- to facilitate association of actions (in motor cortex) to changes in the environment (reinforcement learning)

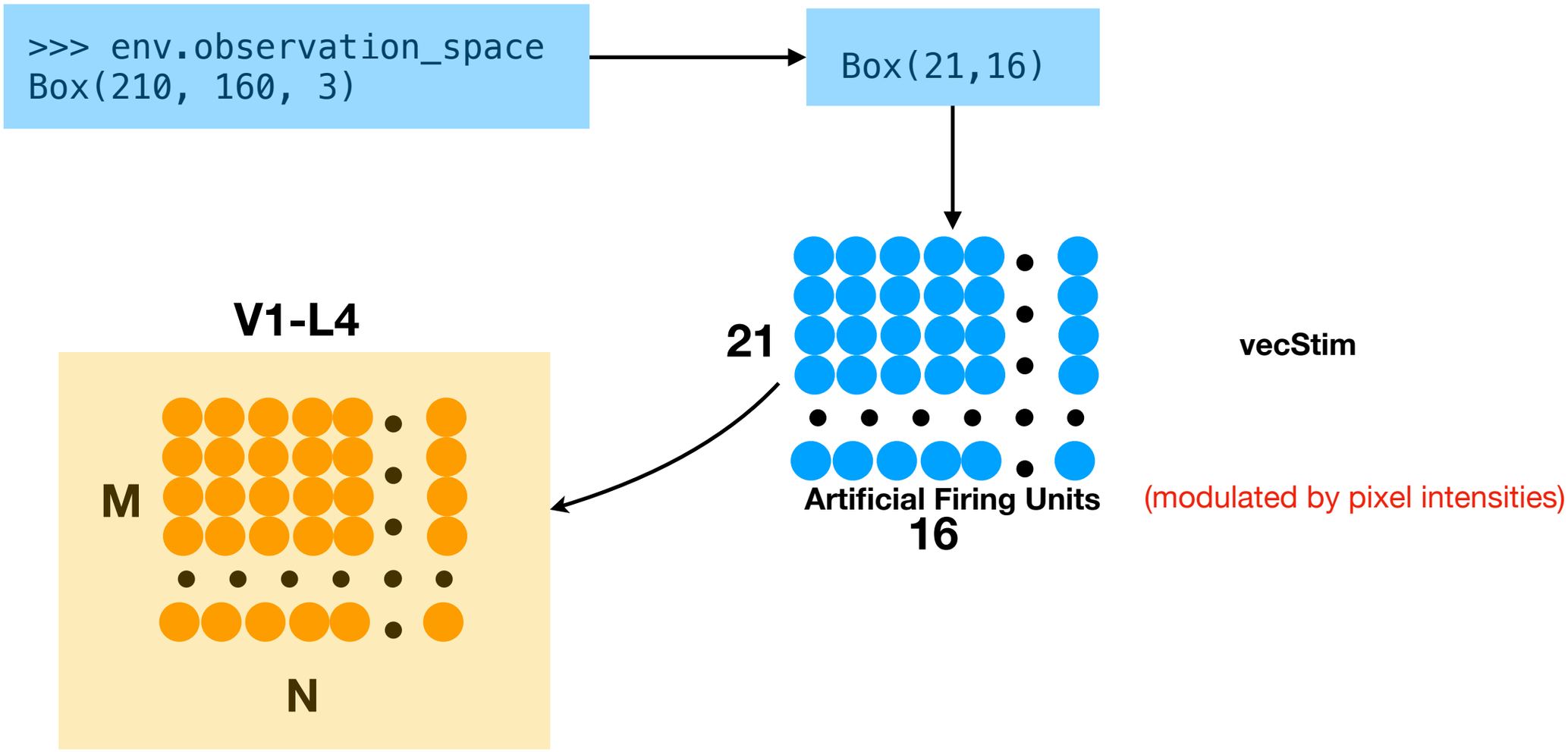



```
python
import random
import gym
env = gym.make("Pong-v0”)
env.reset()

for _ in range(1000):
    env.render()
    action = random.randint(3,4)
    observation, reward, done, info = env.step(action)
    #env.step(env.action_space.sample())
env.close()
```

```
>>> reward
0.0
>>> observation
array([[[  0,   0,   0],
        [  0,   0,   0],
        [  0,   0,   0],
        ...,
        [144,  72,  17],
        [144,  72,  17],
        [144,  72,  17]],
……

>>> env.observation_space
Box(210, 160, 3)
```



How to choose:

what algorithms for unsupervised learning?

- STDP with homeostatic synaptic scaling

what level of details is necessary for Visual Cortex model to encode/interpret visual environment of the game? - **iterative approach**

How to test/validate that the model can learn about the game environment?

- object recognition - e.g. racket, ball
- object associations - e.g. location of racket with respect to ball, speed/velocity

Multi-layer network utilizing rewarded spike time dependent plasticity to learn a foraging task

(Sanda, Skorheim, Bazhenov: 2017- PLoS Comp Biol)

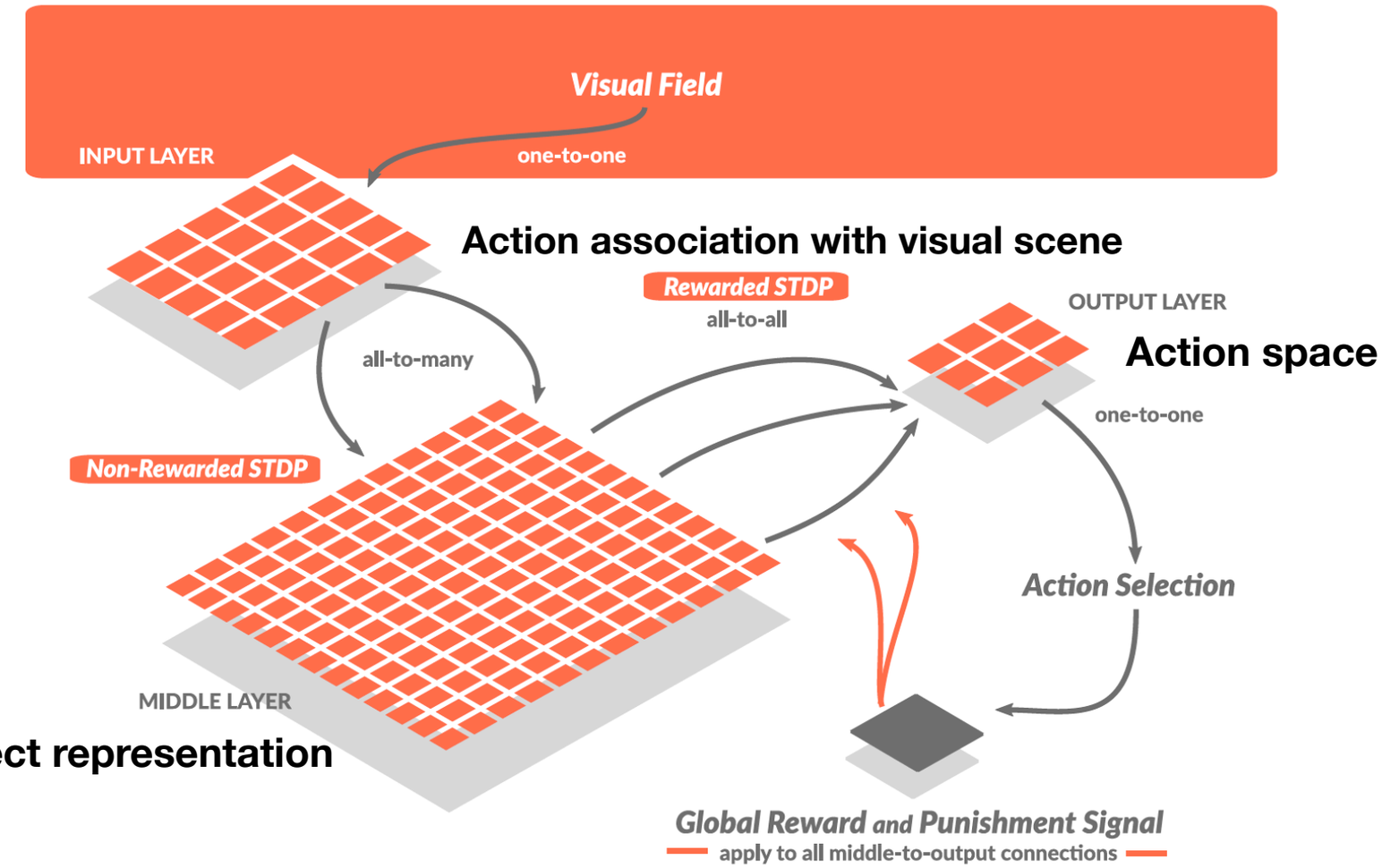


Fig 1. Network organization. The network organization is a simplification of the information processing flow known in the visual pathway, involving mapping of the sensory input into the higher level representations and then using them for decision making in the prefrontal cortex [4]. Input indicating the position of food particles relative to the virtual agent (positioned in the center of the field) was simulated as a set of excitatory inputs to the input layer neurons. In the model, each input layer cell sends one excitatory and one inhibitory connection to each of the cells in the middle layer where object representation is built. Each middle layer cell sends one excitatory and one inhibitory connection to 9 cells in the output layer. The most active cell in the output layer (size 3x3) decides the direction of subsequent movement. Excitatory connections from the input to the middle layer are subject to non-rewarded STDP. Excitatory connections from the middle layer to the output layer are subject to rewarded STDP where reward depends on whenever a move results in food acquisition. Inhibitory connections from a given cell always match the average strength of the excitatory outputs of the same presynaptic cell.

all-to-many: each input layer cell ——> (1 Inh, I exc) to each cell of the middle layer cells

Salvador— —
“so I built this into netpyne such that if you create a population of NetStim and set ‘rate’: ‘variable’ it uses the nsloc.mod mech”

```
netParams.stimSourceParams['stimPsh'] = {'type': 'NetStim', 'rate': 'variable', 'noise': 0} # stim inputs for P_sh

for stim in cell.stims:
        if stim['source'] == 'stimEM':
            stim['hObj'].interval = 1000/self.randRate
            break
```

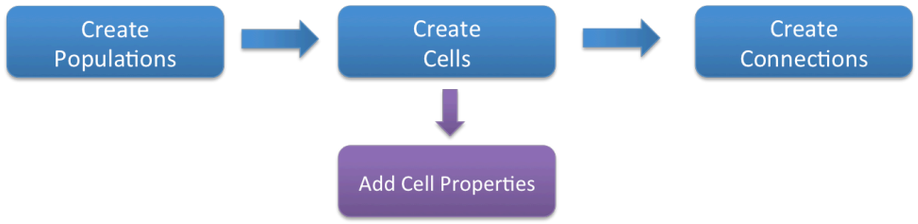
How to access variable NetStim

```
sim.net.cells[7].stims
[{source: 'stimMod', type: 'NetStim', rate: 'variable', noise: 0, number: 1000000000.0, start: 0, seed: 1, hRandom: Random[28],
hObj: NSLOC[3]}]
sim.net.cells[7].stims[0][‘hObj'].interval
```

How to update NetStim at each time step

```
sim.runSimWithIntervalFunc(interval,func)
interval- time in ms, when you want to update the parameters of NetStim
```

STEP 1: create artificial cells generating spikes. The rate of spikes should depend on the input these neurons receive.



1. Trim 210x160 image into 160x160 image (31:190)

Sigmoid function : transform pixel intensity to firing rate

import numpy as np

```
a = np.multiply(0.01,range(0,25500))
b = 40/(1+np.exp((-a+123)/25))

import matplotlib.pyplot as plt
plt.plot(a,b)
plt.xlabel('pixel intensity')
plt.ylabel('firing rate')
plt.show()
```

I thought that DL can only be used for object recognition because i was not sure how can neurons in an input layer encode object displacement across time. This is relevant because I want to understand how DL can be used to train playing games because the environment is constantly changing and in case of pong we will need to estimate the movement of the ball relative to the racket to learn about any reward based actions. The trick often used is by taking the difference between two consecutive image frames and then adding 4 difference frames. It will give motion information that can be used to train DNNs. e.g. from the independent study by Devdhar Patel, the preprocessing is done as described below:

Preprocessing

Each frame from the gym environment is cropped to remove the text above the screen displaying the score and the number of lives left. The image is then re-sized to a 80x80 image and converted to a binary image. The previous frame is then subtracted from the current frame while clamping all the negative value to 0. We then add the most recent 4 difference frames to create a state. Thus, a state is a 80x80 binary image containing the movement information of the last 4 states. Figure 5 shows an example of the input state for the network. The image shows the movement of the ball and the paddle, however, it is not possible to detect the direction of the movement from the image.

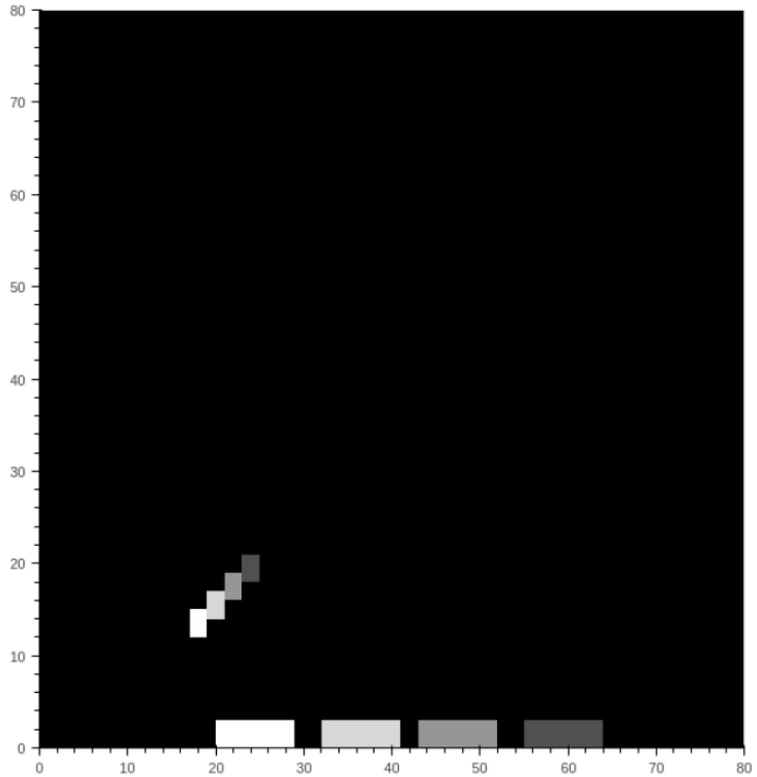


Figure 12: Example grayscale input

To encode direction information into the state/image:

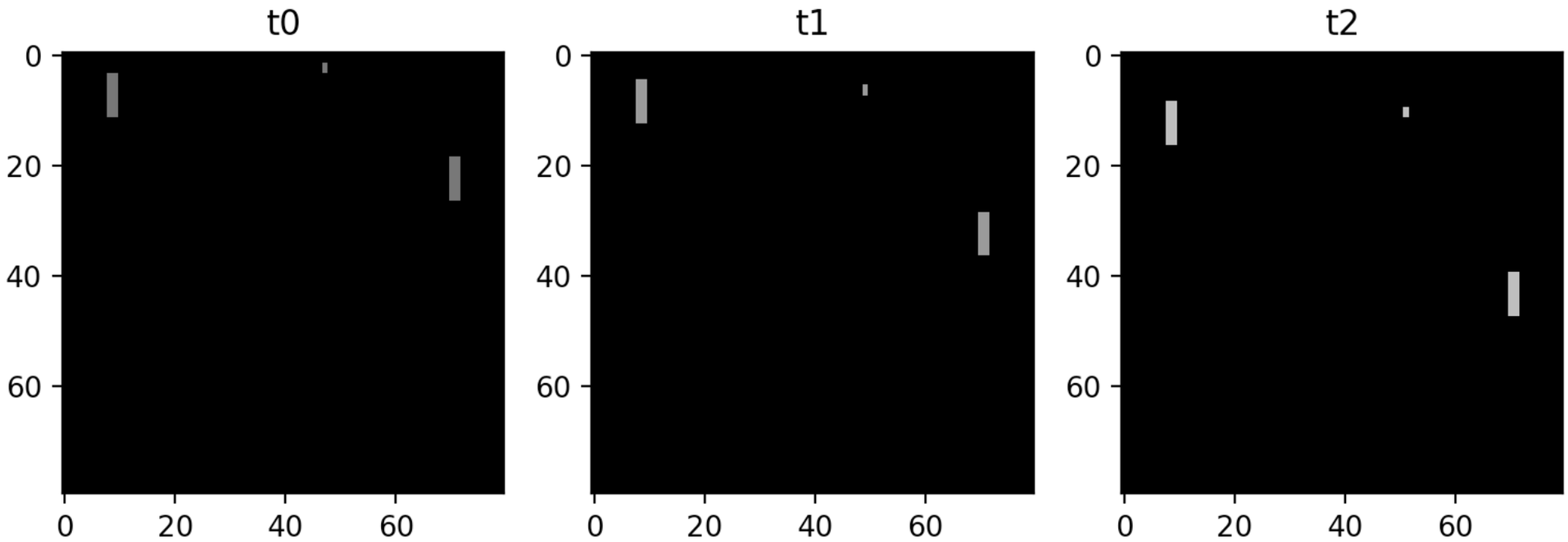
$$S_t = F_t * 1 + F_{t-1} * 0.75 + F_{t-2} * 0.5 + F_{t-3} * 0.25$$

Where S_t and F_t are the state and the frame at time t respectively.

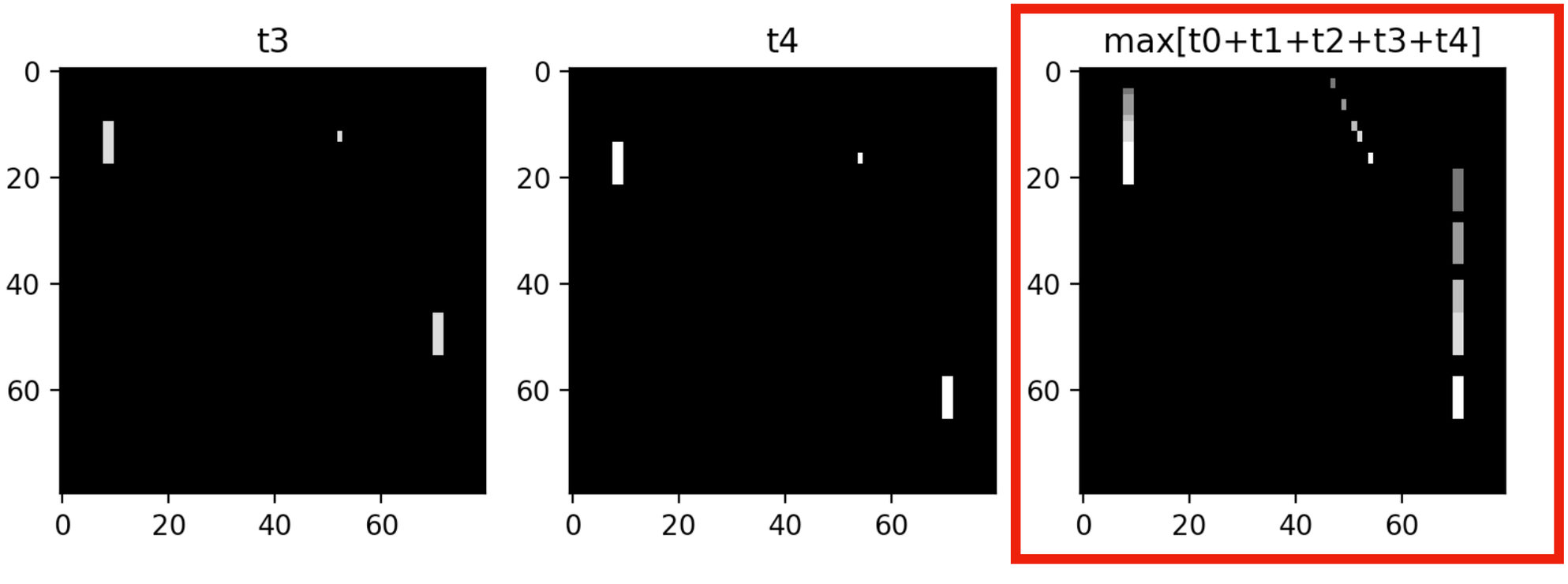
I slightly modified the approached as described below:

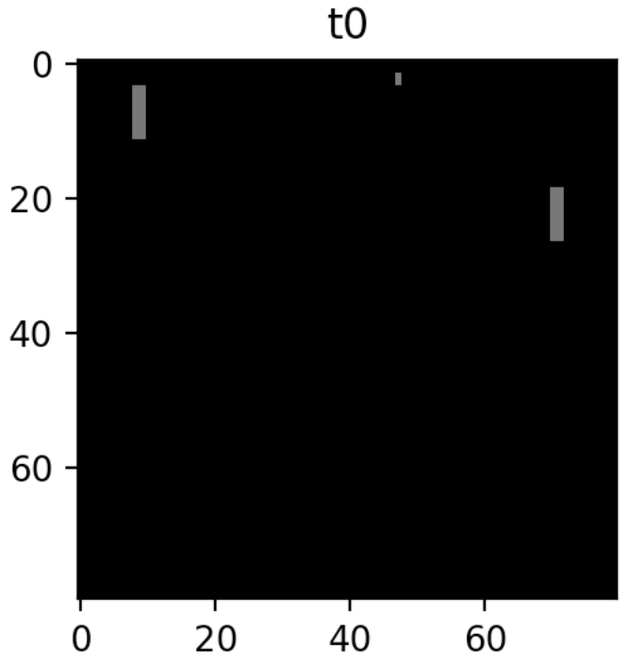
Model Input: *Image showing trajectory of ball and rackets*

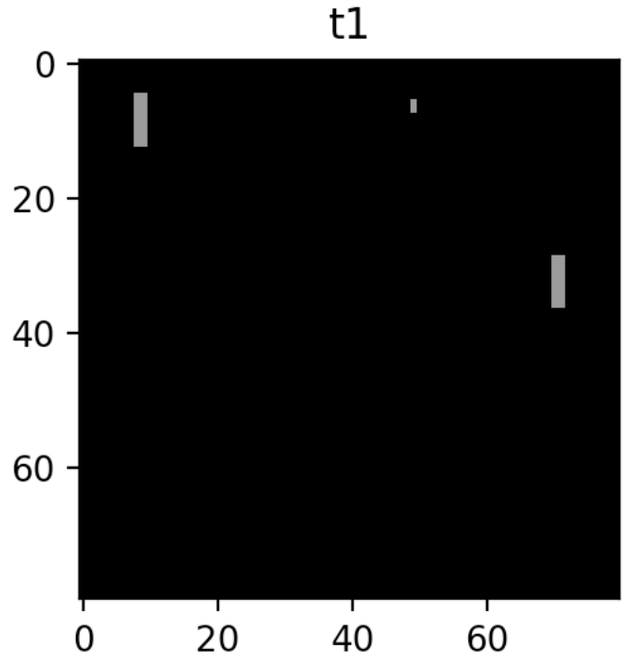
1 input image after 5 actions in the game

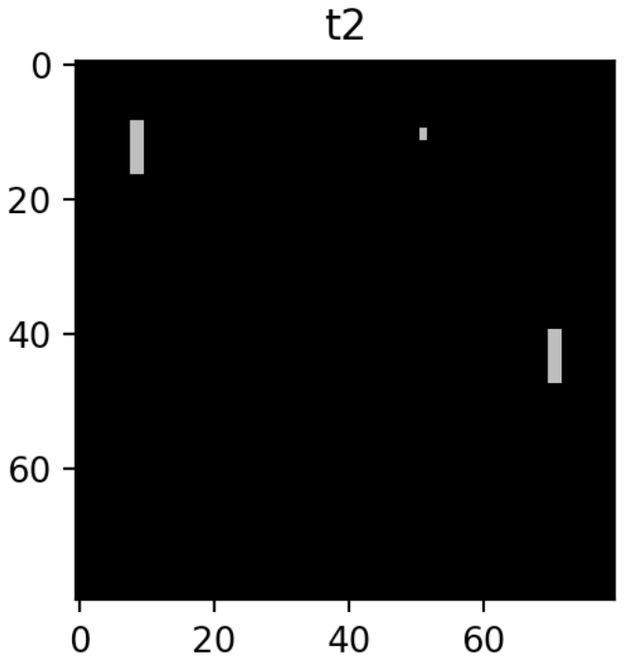


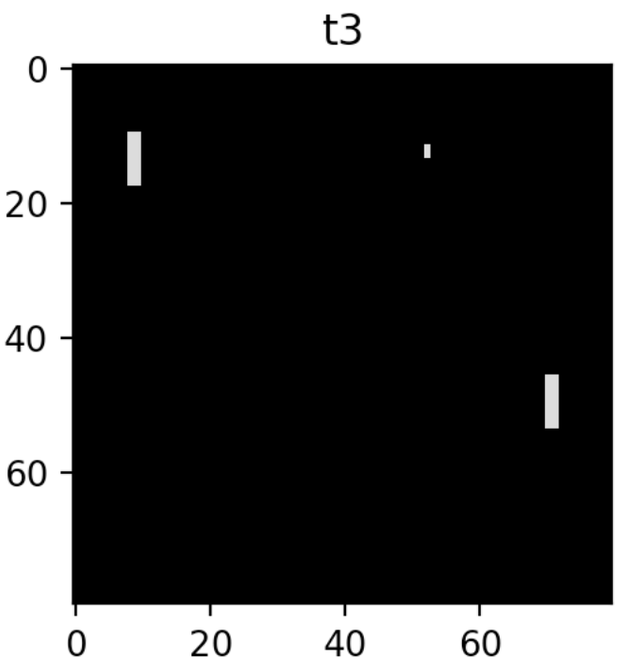
send to the network at every 20 ms

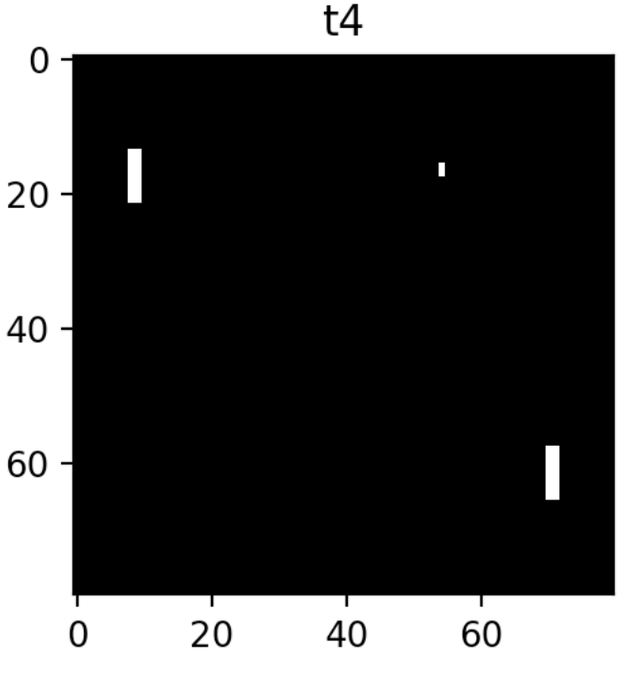


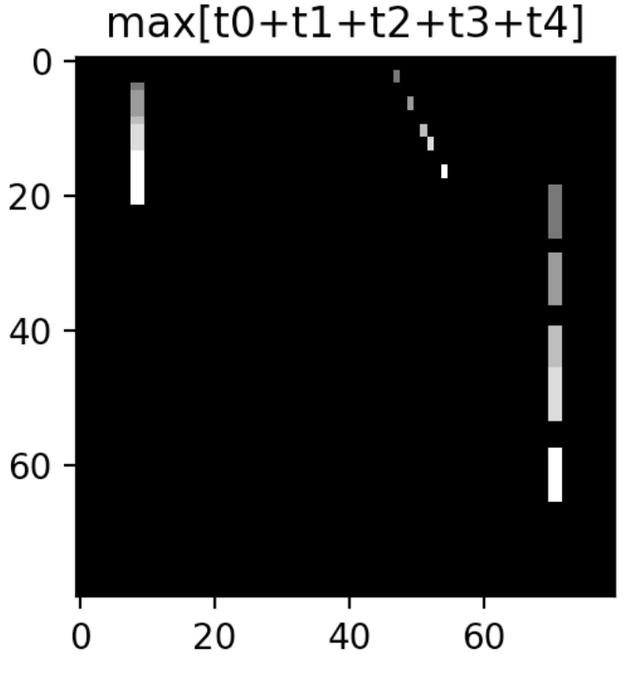

t0
0
20
40
60
0
20
40
60


t1
0
20
40
60
0
20
40
60

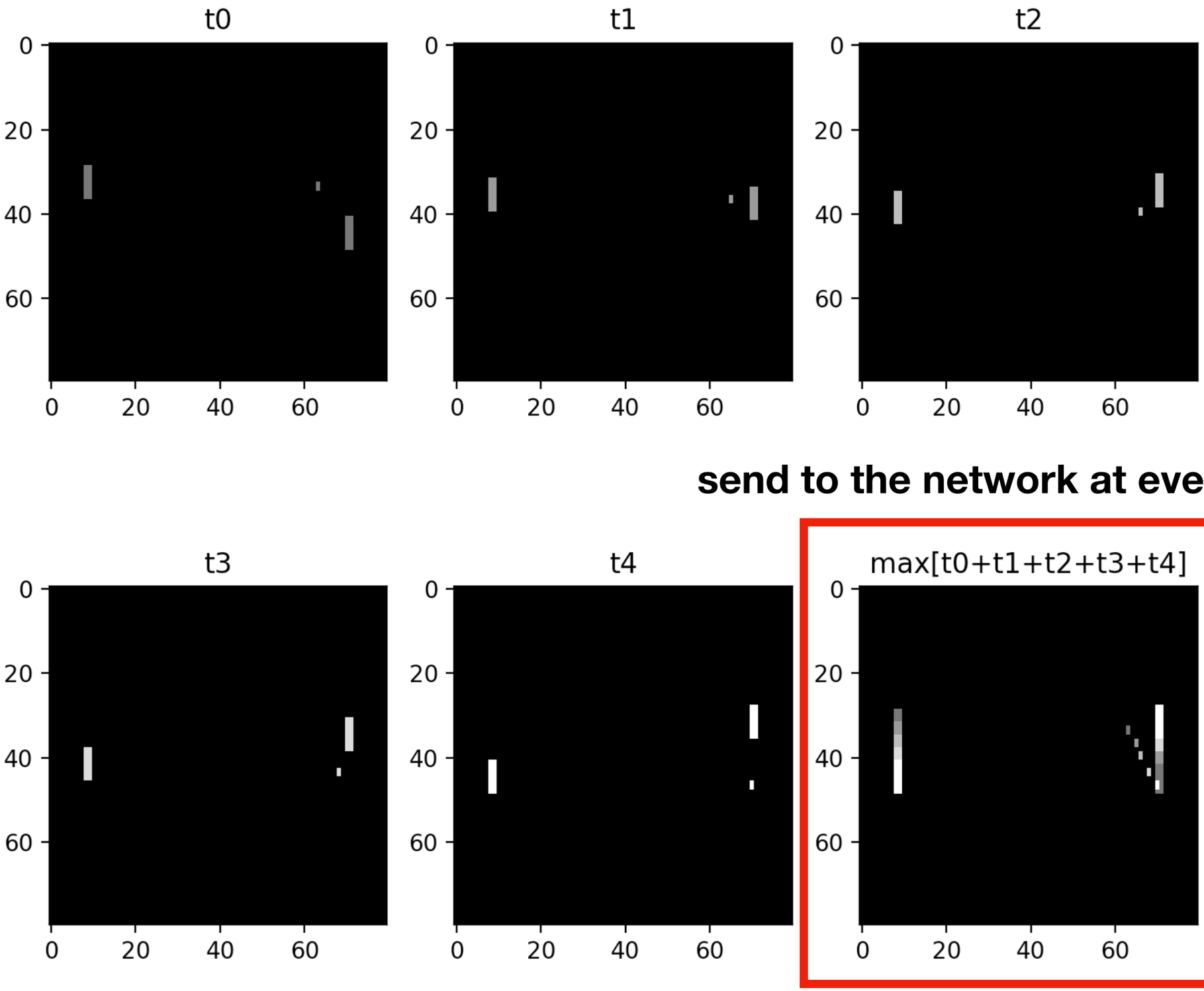

t2
0
20
40
60
0
20
40
60


t3
0
20
40
60
0
20
40
60


t4
0
20
40
60
0
20
40
60

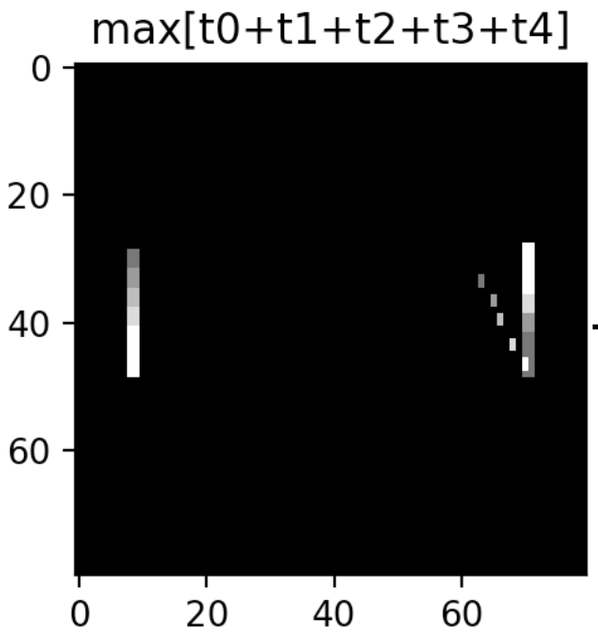

max[t0+t1+t2+t3+t4]
0
20
40
60
0
20
40
60

Model Input: *Image showing trajectory of ball and rackets*



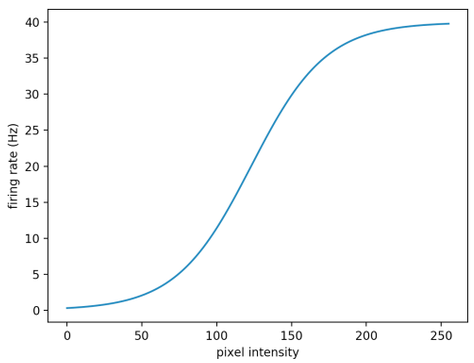
send to the network at every 20 ms

Model Input —> Spike generation



80x80 pixels

1 pixel :1 stim



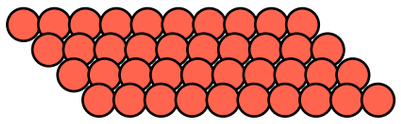
6400 Artificial neurons

-

Stim(variable firing rate)

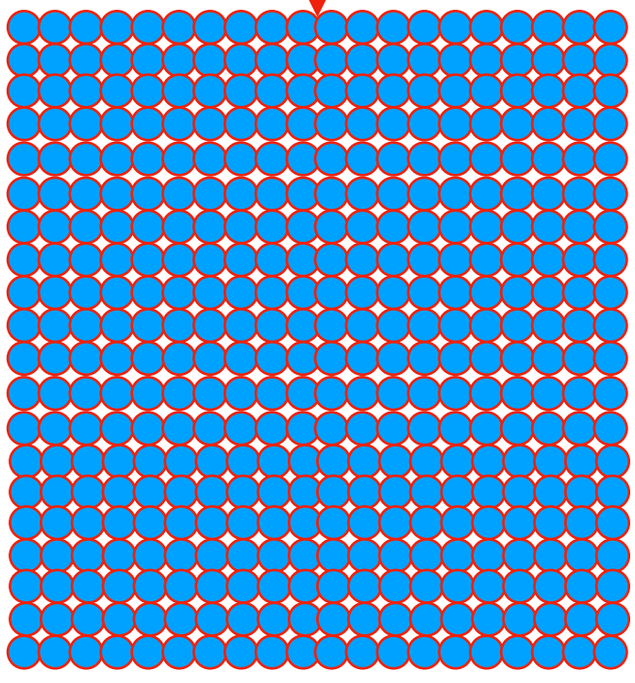
1:1 (fixed weight)

0.007



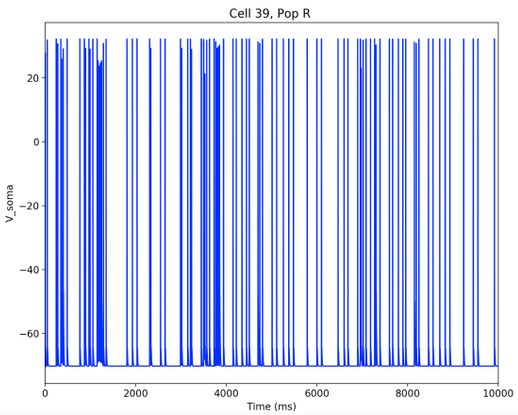
Add topographic details to mapping fro R to V1

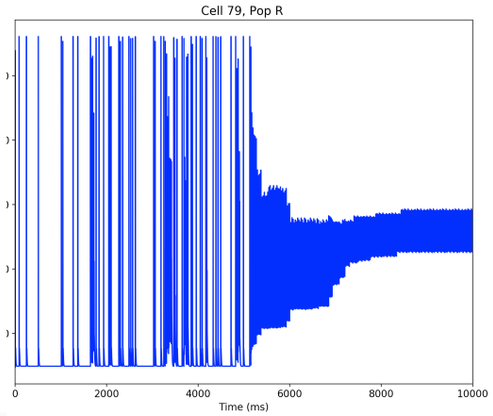
different sizes - test - convergence

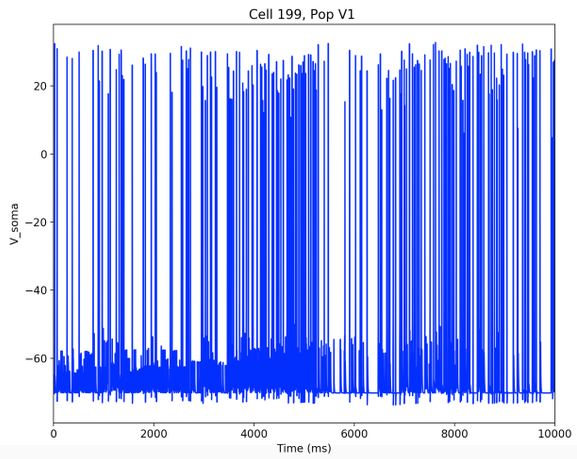


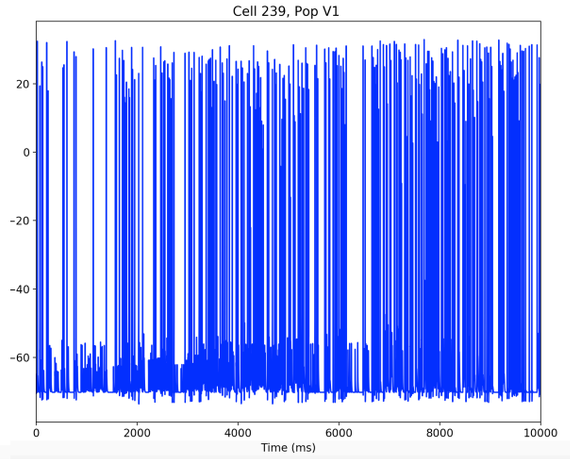
6400 HH

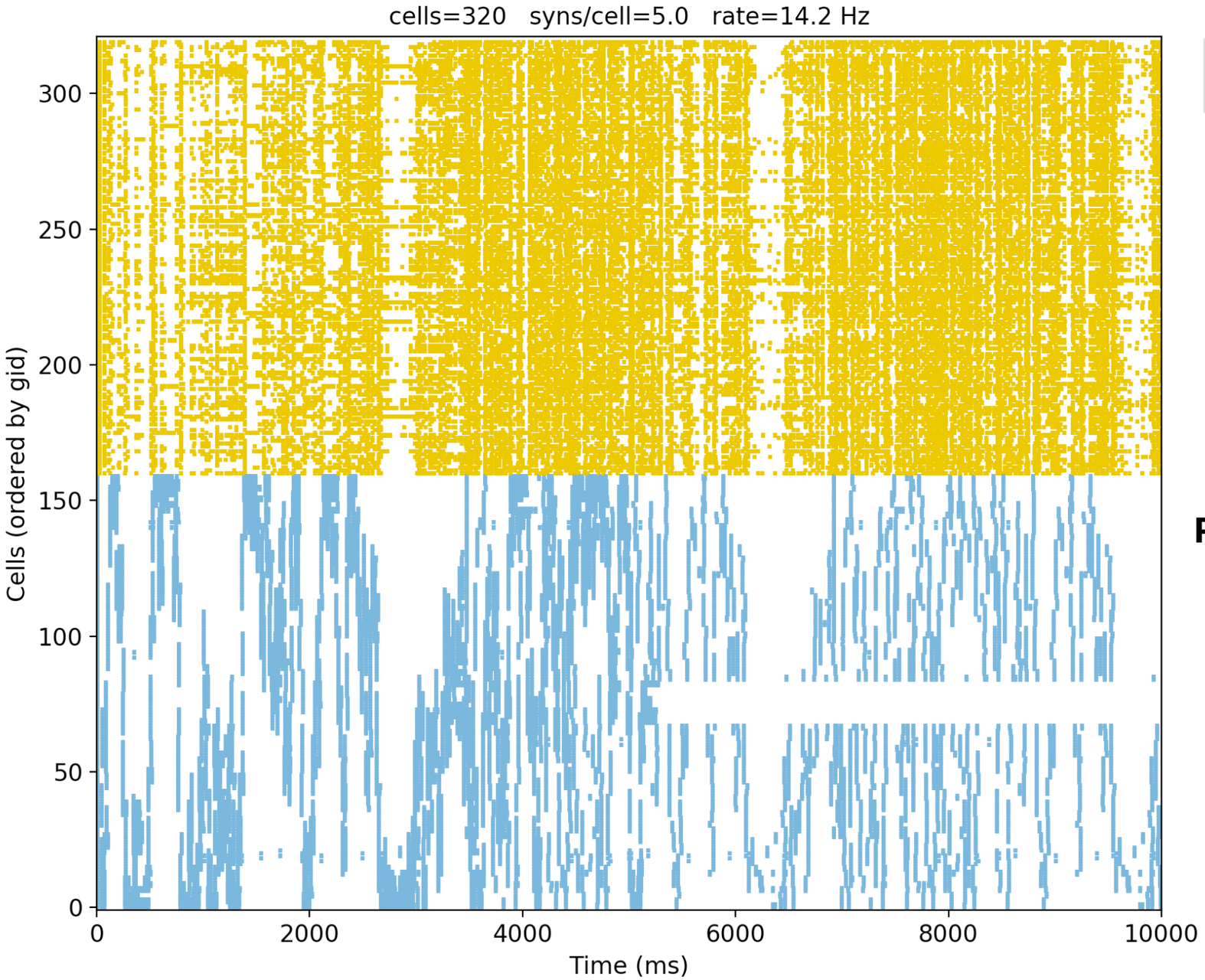
spiking neurons

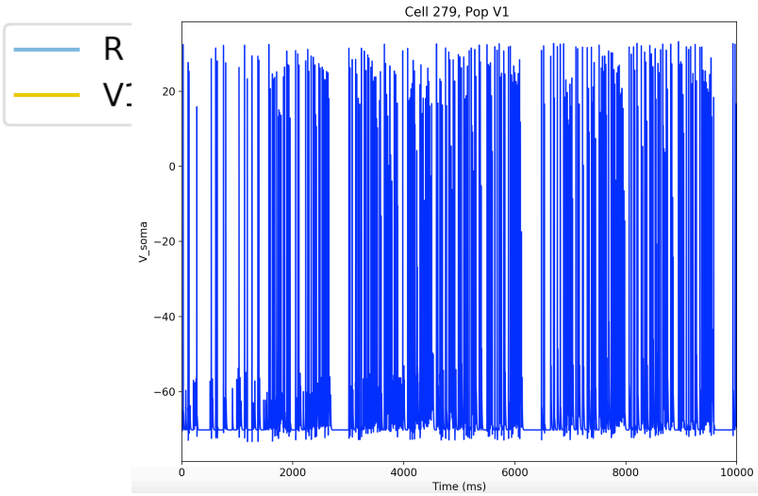












Problem: Some neurons stop firing

possible reasons:

1. too strong synaptic strength
2. very large input from Stim cells

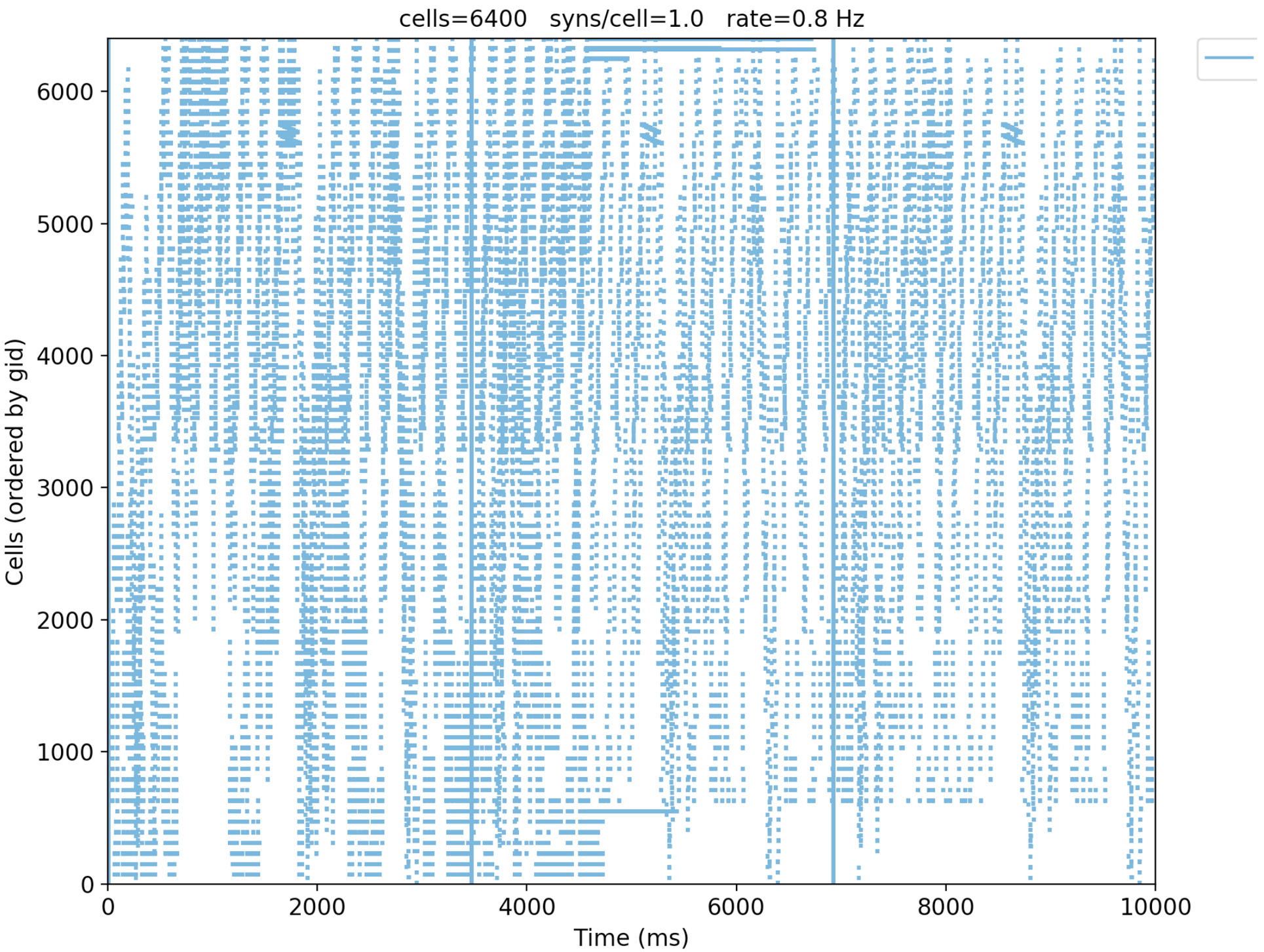
The reason is that the game stops and therefore the neurons keep getting the same input for rest of the simulation

Fixed: reset the game environment after 1000 repeats

Interfacing game env with NetPyNe

- We play 5 moves/actions in pong game and use those steps to construct an image frame.
- We allow T1 msec to Neural Network Model (SMARTAgent) to represent the image frame and then again play 5 moves/ actions.
- I think that brain needs at least 20-40 ms to detect the visual input therefore I choose T1 = 20 msec.
- delay of excitation from input layer is 1 ms and ‘weight’ is 0.01
- when i use ‘weight’ of 0.005, some Stims produce spikes, while others fail to produce spikes.
- when i use ‘weight of 0.001’ only Stims with very high frequency could produce a few spikes

For now I am using 1:1 input mapping—-> I may change this to —-> 16 pixels mapping onto 1 neuron



What range of ‘weights’ would allow optimal transfer of information?

- As noticed in simulations run using different ‘weights’ of Stims connecting a layer of neurons that:
    - weight of 0.001 was so low that only Stims at higher frequencies could evoke spikes in the post-synaptic neurons.
    - weight of 0.01 was high enough so that (possibly, not all neurons checked) all Stims could evoke spikes in post-synaptic neurons.
    - increasing weight over 0.01 reduced spiking rate of the network.

Need to be careful here, because increasing weights beyond a range may not allow transfer of spiking rate across layers of neurons

‘weights’	0.006	0.007	0.0075	0.008	0.01
Trial 1	10.3	10.6 Hz	9.5	10.3	10.3
Trial 2	10.0	9.2 Hz	10.5	9.8	7.9
Trial 3	10.1	11.7 Hz	10.2	10.3	9.1
Trial 4	10.1	11.3 Hz	10.0	8.6	9.3
Trial 5	10.3	10.5 Hz	11.3	8.9	8.7

Network structure

The network was composed of 842 spiking map based neurons [97, 98], arranged into 3 feed forward layers to mimic a basic biological circuit: a 7 by 7 input layer (I), a 28 by 28 middle (hidden, H) layer, and a 3 by 3 output layer (O) (Fig 1). This structure provides a basic feedforward inhibitory circuit [73] found in many biological structures [99, 73, 100, 101, 102, 103].

Each cell in the middle layer received synaptic inputs from 9 random cells in the input layer. These connections initially had random strengths drawn from a normal distribution. Each cell in the excitatory middle layer (cell H_i) connected to every cell in the output layer (O_j) with synaptic strength W_{ij} or WI_{ij}, respectively. Initially all these connections had uniform strengths and the responses in the output layer were due to the random synaptic variability. Random variability was a property of all synaptic interactions between neurons and it was implemented as variability in the magnitude of the individual synaptic events.

How to build the architecture?

https://towardsdatascience.com/a-guide-to-an-efficient-way-to-build-neural-network-architectures-part-ii-hyper-parameter-42efca01e5d7

A convolution of pooling layer reduces the dimensions of the input image of dimension N to (N-f+1)/s where ‘f’ is the filter size and ’s’ the stride length (stride controls how the conv. filter slides on input e.g. with stride length of 1, it will move pixel by pixel.)

Make a sheet containing the dimensions, activation shapes and sizes for the architecture.

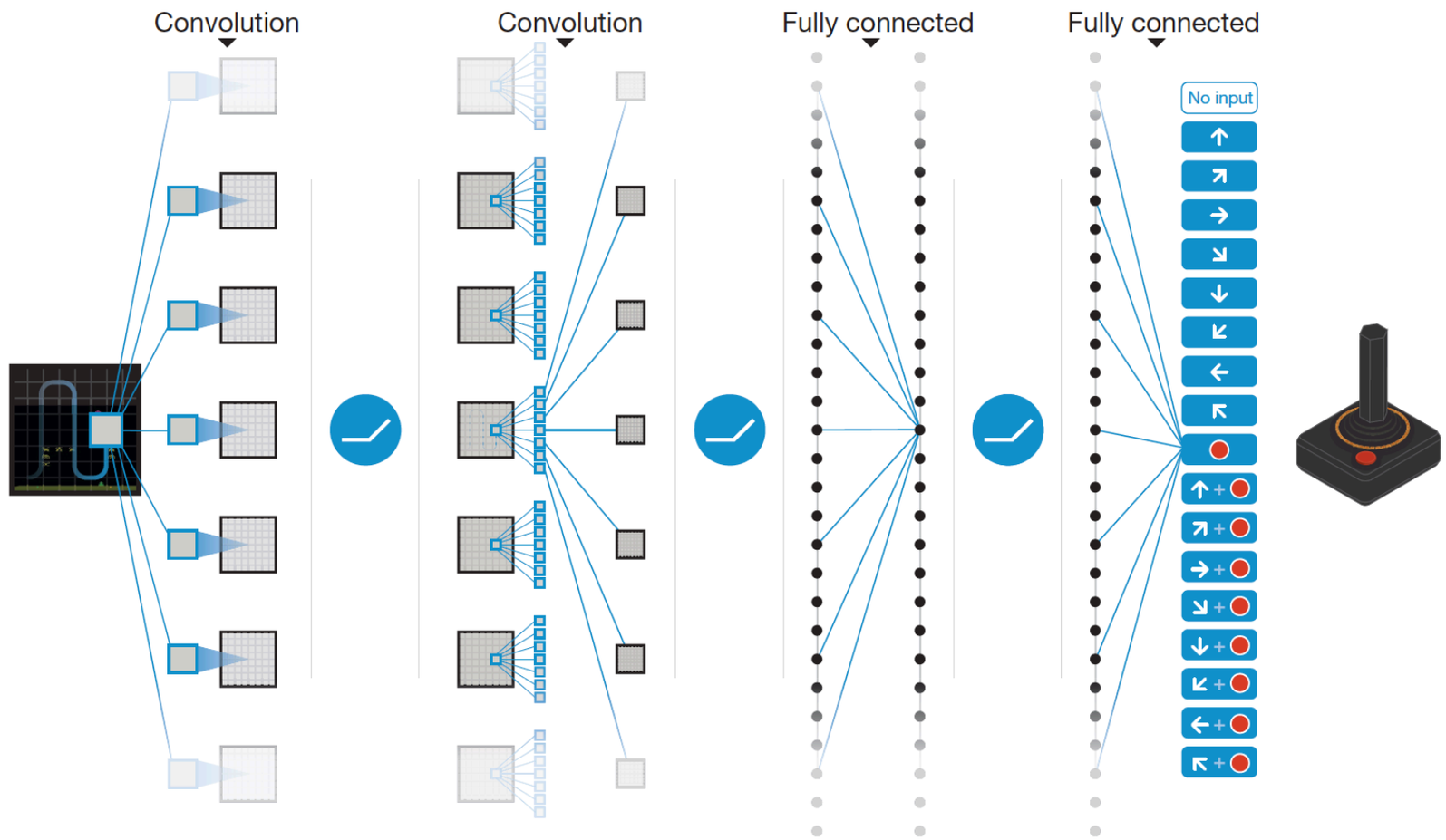


Figure 1 | Schematic illustration of the convolutional neural network. The details of the architecture are explained in the Methods. The input to the neural network consists of an $84 \times 84 \times 4$ image produced by the preprocessing map ϕ, followed by three convolutional layers (note: snaking blue line symbolizes sliding of each filter across input image) and two fully connected layers with a single output for each valid action. Each hidden layer is followed by a rectifier nonlinearity (that is, $\max(0,x)$).

The exact architecture, shown schematically in Fig. 1, is as follows. The input to the neural network consists of an 84 × 84 × 4 image produced by the preprocessing map ϕ. The first hidden layer convolves 32 filters of 8 × 8 with stride 4 with the input image and applies a rectifier nonlinearity[31,32]. The second hidden layer convolves 64 filters of 4 × 4 with stride 2, again followed by a rectifier nonlinearity. This is followed by a third convolutional layer that convolves 64 filters of 3 × 3 with stride 1 followed by a rectifier. The final hidden layer is fully-connected and consists of 512 rectifier units. The output layer is a fully-connected linear layer with a single output for each valid action. The number of valid actions varied between 4 and 18 on the games we considered.