Braindrop Pystorm Documentation

This documentation follows along with sections of test_hal_decode.py. Follow along in that code to make sure everything makes sense!

Table of Contents

Network Objects

A pystorm network is made up of **inputs**, **buckets**, **pools**, **connections**, and **outputs** (**Figure 1**).

Inputs and outputs are kinda self-explanatory; they're data streams that either come on to the chip from the host, or go back to the host from the chip.

Pools contain the spiking neurons themselves. Pools are defined by their x and y location and size in the neuron array, their tap point matrix, their biases, their gain divisors, and their diffuser cuts.

The biases and gain divisors allow us to get neurons that would either be always-on or always-off into a range where they can be usable. The allowed gains are 1x, 1/2x, 1/3x, 1/4x, while the allowed biases are -3, 2, 1, 0, 1, 2, 3. Gains and biases can each be set as constant across a pool, or each be defined as an array of length N, where N is the number of neurons.

The diffuser is equivalent to a resistive network that allows synapse outputs to spread across many neurons. Each synapse always is connected to 4 neurons, but can be connected to additional neurons through the diffuser. By default, there are no diffuser cuts.

Buckets allow us to decode spikes coming from pools. Buckets accumulate their inputs, usually a spike train from a pool, weighting these by some decoder. Every time a bucket's ±1 threshold is crossed, it outputs a ±1 event, addressed with a tag. That tag corresponds to an entry in the Tag Action Table (TAT) that tells the chip where that event should be sent: To other buckets, to synapses, to another Braindrop, or to the host PC.

Connections allow us to connect elements. A connection can have weights, represented by a decoder matrix, only if the destination of the connection is a bucket. The decoder matrix is multiplied by the connection's source vector to yield its destination vector (using buckets).

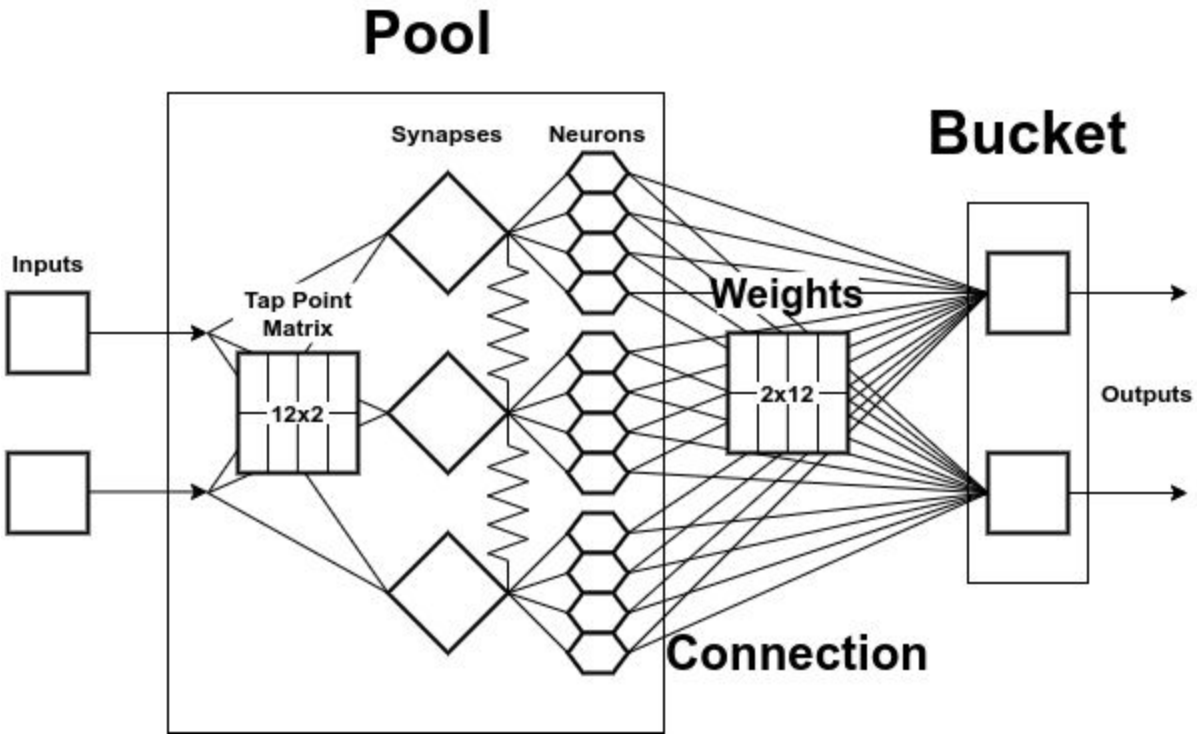
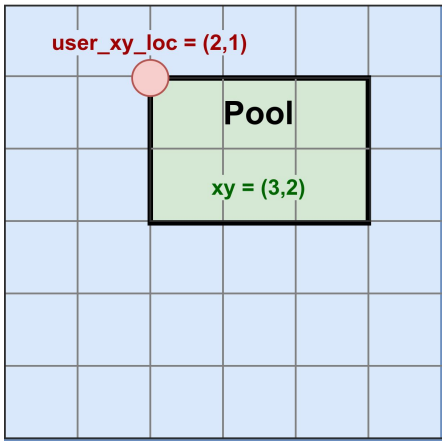


Figure 1: Abstractions you use to program Braindrop with Pystorm.

Pool

Function Header:
 	
`def create_pool(self, label, taps,`
`gain_divisors=1, biases=0,`
`xy=None, user_xy_loc=(None, None),`
`diffusor_cuts_yx=None,`
`allow_weird_taps=False):`

Parameters: Label (string): The name of the pool. Taps (array or tuple): The tap-point matrix. See below. Gain_divisors (array or int, *optional*): Divide the input gain of a neuron. The allowed divisors are 1,2,3, and 4, corresponding to gains of 1, ½, ⅓, and ¼ . If an int is passed in, it's expanded to an array with that gain divisor for every neuron. Biases (array or int, *optional*): Bias the input of the neuron. The allowed biases are -3, 2, 1, 0, 1, 2, 3. If an int is passed in, it's expanded to an array with that bias for every neuron. xy (tuple, *optional*): The shape of the pool, in neurons, as (x,y). User_xy_loc (tuple, *optional*): The location of the pool, in neurons, as (x,y). The origin, (0,0), is located at the top left of the neuron array, and user_xy_loc describes the position of the top left corner of the pool of neurons. **Neuron Array**   

Diffusor_cuts_yx (list, *optional*): A list with tuple elements (y, x, direction), indicating where the diffuser is cut. Both x and y should be relative to the top left corner of the pool. Possible directions are “right”, “left”, “up”, and “down”. The edges of the pool are cut automatically, to prevent currents from diffusing across the pool edges. The diffuser can only be cut at edges of a 4x4 tile of neurons.

Example

```
tap_matrix = np.zeros((64, 1)) # 64 neurons, 1 dimension
width = 8
height = 8
# one synapse per 4 neurons
for x in range(0, width, 2):
     for y in range(0, height, 2):
        n = y * width + x
        if x < width // 2:
            tap_matrix[n, 0] = 1
        else:
            tap_matrix[n, 0] = -1

# cut the diffuser everywhere
# diffuser can only be cut at at 4x4 tile edges
diff_cuts = []
for x in range(0, width, 4):
     for y in range(0, height, 4):
        diff_cuts.append((y, x, “right”))
        diff_cuts.append((y, x, “left”))
        diff_cuts.append((y, x, “up”))
        diff_cuts.append((y, x, “down”))

# create an 8x8 neuron pool located at (16,16)
p1 = net.create_pool("p1", tap_matrix, gain_divisors=2, biases=-3,
xy=(8,8), user_xy_loc=(16, 16), diffusor_cuts_yx=diff_cuts)
```

Bucket
Function Header: `def create_bucket(self, label, dimensions):`
Parameters: Label (string): The name of the bucket. Dimensions (int): The number of dimensions the bucket has.
Example `b1 = net.create_bucket("b1", Dout)`

Input
Function Header: `def create_input(self, label, dimensions):`
Parameters: Label (string): The name of the input. Dimensions (int): The number of dimensions the input has.
Example `i1 = net.create_input("i1", Din)`

Output

Function Header:

```
def create_output(self, label, dimensions):
```

Parameters:

Label (string): The name of the output.

Dimensions (int): The number of dimensions the output has.

Example

```
o1 = net.create_output("o1", Dout)
```

Connection

Function Header:

```
def create_connection(self, label, src, dest, weights):
```

Parameters:

Label (string): The name of the connection.

Src (network object): The object this connection is coming from.

Dest (network object): The object this connection is going to.

Weights (array): A matrix, sized (Dimensions of destination)x(Dimensions of source). The connection multiplies the values from the source by this matrix. Only connections where the destination is a Bucket can have weights.

Example

```
decoders = np.zeros((Dout, N))
net.create_connection("c_i1_to_p1", i1, p1, None)
d_conn = net.create_connection("c_p1_to_b1", p1, b1, decoders)
net.create_connection("c_b1_to_o1", b1, o1, None)
```

In test_hal_decode.py, we use one pool and one bucket to perform our decoding. Our decoder is of the dimensions Dout x N, where Dout is our output dimensionality and N is the number of neurons in our pool.

```
##########################################
# specify network using HAL

net = graph.Network("net")

# decoders are initially zero, we remap them later (without touching
the rest of the network) using HAL.remap_weights()
decoders = np.zeros((Dout, N))
#decoders = np.ones((Dout, N)) * .2 # sanity check: is accumulator
mapping correctly?

i1 = net.create_input("i1", Din)
p1 = net.create_pool("p1", tap_matrix, biases=-3)
b1 = net.create_bucket("b1", Dout)
o1 = net.create_output("o1", Dout)

net.create_connection("c_i1_to_p1", i1, p1, None)
decoder_conn = net.create_connection("c_p1_to_b1", p1, b1, decoders)
net.create_connection("c_b1_to_o1", b1, o1, None)
```

Tap Points

We use tap points to specify which pool inputs go to which synapses. These connections can be positive (+1), absent (0), or negative (-1). They can be formatted densely or sparsely using:

1. A matrix (pre-diffuser), size neurons-by-dimensions.
   Elements must be in {-1, 0, 1}.
   Implicitly describes pool dimensionality and number of neurons.
2. A list (N, [[tap dim 0 list], ... , [tap dim D-1 list]])
   Elements must be (neuron idx, tap sign), where tap sign is in {-1, 1}

Note: A tap-point matrix is sized neurons-by-dimensions, but actually corresponds to which synapse gets which pool input, not which neuron. As there are four neurons per synapse, only one of each set of four adjacent neurons should have a non-zero tap-point assigned to it (otherwise, errors get thrown).

In test_hal_decode.py, we split the tap points, so half of the neurons have a tap point of 1 and the others get -1.

```
##########################################
# tap point specification

tap_matrix = np.zeros((N, Din))
if Din == 1:
      # one synapse per 4 neurons
      for x in range(0, width, 2):
      for y in range(0, height, 2):
            n = y * width + x
            if x < width // 2:
                  tap_matrix[n, 0] = 1
            else:
                  tap_matrix[n, 0] = -1
else:
      print("need to implement reasonable taps for Din > 1")
      assert(False)
```

DAC Values

Braindrop has multiple global DAC values that allow the user to change the behavior of the chip's neurons in various ways. These DAC values can be queried with the function:

```
hal.get_DAC_value(dac_name)
```

They can be set with the function:

```
hal.set_DAC_value(dac_name, value)
```

A table of DAC names and their functions is provided below. Each DAC can be set to a value between 1 and 1024.

DAC Name	**DAC Function**	**Default Values**	**Scaling**
DAC_SYN_EXC	Synaptic excitatory level	512	8
DAC_SYN_DC	Synaptic DC baseline	544	16
DAC_SYN_INH	Synaptic inhibitory level	512	128
DAC_SYN_LK	Synaptic leak (sets time constant)	10	160
DAC_SYN_PD	Synapse Pulse Extender fall time	40	1
DAC_SYN_PU	Synapse Pulse Extender rise time	1024	1
DAC_DIFF_G	Conductance of the diffuser, upwards and downwards	1024	1
DAC_DIFF_R	Conductance of the diffuser, left-to-right	1024	1
DAC_SOMA_REF	Soma refractory period	10	1
DAC_SOMA_OFFSET	Soma offset (scales the bias twiddle bits)	2	4

Disabling Neurons and Synapses: "Kill Bits"

Neurons and synapses that are performing badly (for example, a neuron that is always on, or a synapse that's always feeding current to its neurons) can be disabled to significantly increase output performance. A soma at position x,y can be disabled using:

```
hal.driver.DisableSomaXY(CORE_ID, x, y)
```

A synapse at position x,y can be disabled using:

```
hal.driver.DisableSynapseXY(CORE_ID, x, y)
```

For all single-braindrop systems, CORE_ID is always zero.

Tau-Calibrated Pools

Nengo-brainstorm has the ability to automatically calibrate pools. (This functionality is still under development, in the “cal_ens_attempt” branch of the nengo-brainstorm repository). This generalizes mapping of dynamical systems to mismatched synapses.
It works by first gathering the data on the different tau values from the chip and loading them into the calibration database. This has already been done for the chip currently in use on the ng-amygdala server, but (for reference) can be done like this in case the chip needs to be recalibrated.

```
cd ~/pystorm/pystorm/calibration
python3 add_chip_to_cal_db.py skull
python3 syn_bias_mismatch.py
python3 syn_tau.py 1 --num_trials=25
```

This set of commands collects the time constants with the synapse leak value set to 1.

Then, synapses are clustered based on their time constants, so that synapses with similar time constants are in the same cluster. Each cluster provides a +1 and a -1 tap point for each input dimension in the pool; thus each cluster contains 2D synapses, where D is the number of input dimensions. This forms a tap point matrix with T nonzero tap points, where T = 2DC, where C is the number of clusters used.

To calibrate the input and feedback connections, we need to build a calibration matrix. This matrix is of the size D by (T/2), where D is the input dimensionality and T is the number of tap points used. For each tap-point pair, the input value for its corresponding input dimension is multiplied by the average time constant of its cluster and sent to a set of input buckets, which is of size T/2 = DC; these buckets are connected to the underlying pool using the tap-point matrix described above.

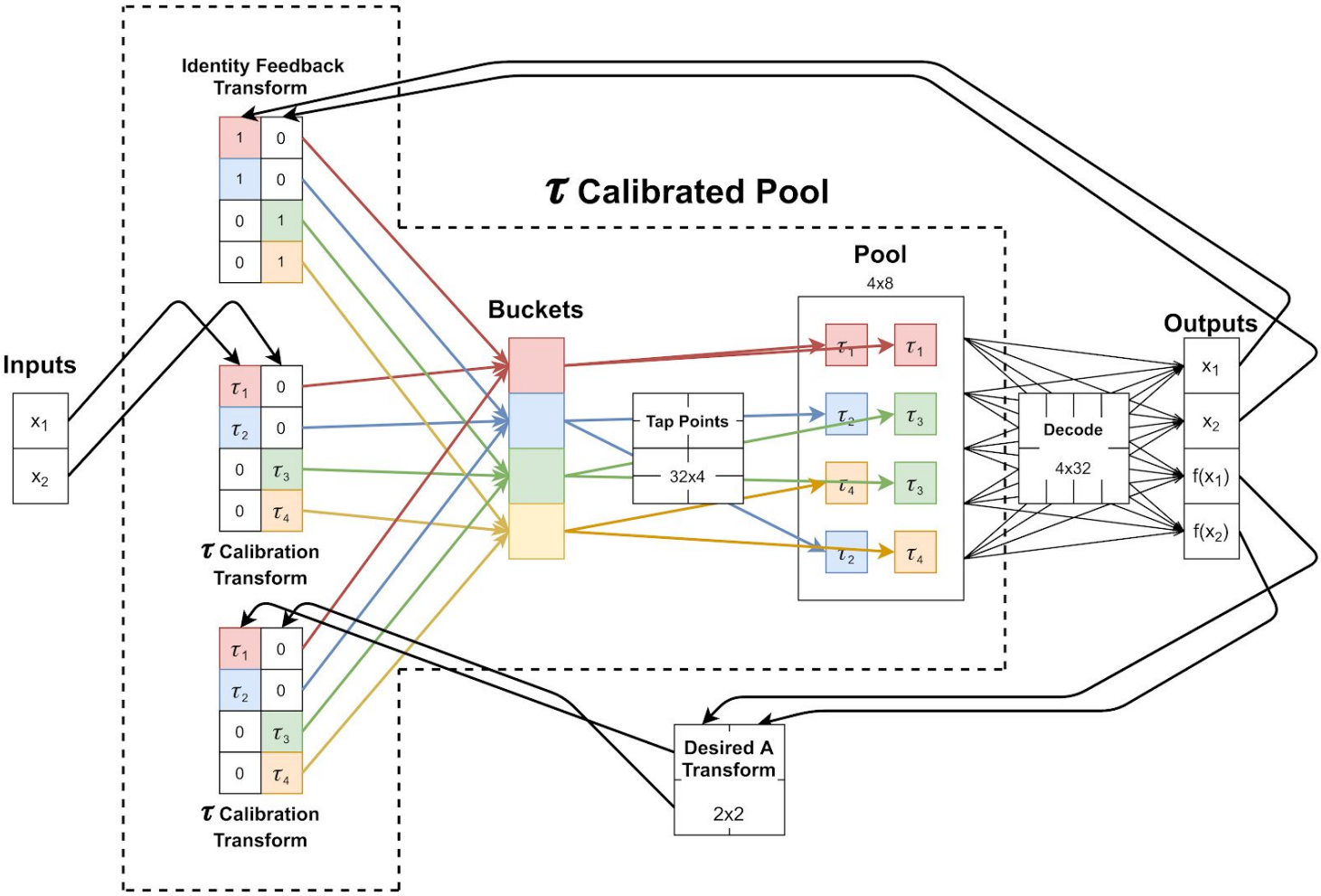
We also need to have an identity feedback connection to compensate for the leak. Thus, out of our pool, we not only decode our nonlinear function of x, but we also decode x itself. Then, we have an additional feedback connection from our identity decode that feeds into the same set of buckets, but for each tap-point pair, multiplies the corresponding dimension of x by 1, rather than by that cluster’s average time constant.

In practice, this means that nengo-brainstorm will automatically calibrate your pools. This means that, when defining nengo-brainstorm pools, you don’t need to multiply B by tau and replace A with tau A + I; this is handled for you, under the hood.

The tau calibration transform for a calibrated pool can be accessed using:

```
tau_cal_transform = sim.cal_transforms[ens]
```

This calibration structure looks like this:



(For a more in-depth explanation of how this works, as well as calibration methods for higher-order taus, see https://web.stanford.edu/group/brainsinsilicon/documents/avoelker.pdf)

Running Input Sweeps and Binning:

Various calls to the **Hardware Abstraction Layer** are required to run the chip:

- **HAL.map()** maps the network to the chip
- **RunControl(HAL, network)** creates a RunControl object
  It uses **set_input_rates** to send input spike rates to the chip.
  It wraps together other lower-level HAL calls to communicate with it properly.

The **run_input_sweep()** call sweeps the pool's inputs to collect spike counts, their bin times, and outputs. This allows the script to determine the necessary tuning curves required to solve for a set of decoders using least squares.

Input values are formatted at a dictionary, with each network input as a key. The value for each network input is a tuple of two numpy arrays; the first contains the input times, while the second contains the input values.
run_input_sweep() returns binned outputs, either from a decode, or raw spikes (chosen using the **get_raw_spikes** parameter. The binning is based on the downstream time resolution passed to the HAL. Most of the time, we want output rates rather than binned outputs; we use **data_utils.bins_to_rates()** to convert our binned outputs to either spike rates, for raw spikes, or output rates, for decoded outputs.

```
##########################################
# invoke HAL.map(), make tons of neuromorph/driver calls under the
hood

# map network
print("calling map")
HAL.map(net)

##########################################
# compute sweep bins

rc = RunControl(HAL, net)

# run an input sweep and collect the spikes & bin times
_, spikes_and_bin_times = rc.run_input_sweep(input_vals,
get_raw_spikes=True, end_time=times_w_end[-1], rel_time=False)

spikes, spike_bin_times = spikes_and_bin_times
# spikes is a dictionary of arrays of the spike counts for each time
bin for the entire chip, indexed by pool
```

```
# convert spike counts per bin to spike rates, make the tuning curve
matrix
A = data_utils.bins_to_rates(spikes[p1], spike_bin_times,
times_w_end, init_discard_frac=.2)
A = A.T

# collect network outputs to flush them from the chip
outputs = HAL.get_outputs()
print("got outputs, doing binning")
```

Using this, we can then determine what our decoder should be to allow us to successfully decode our desired function. We start by dividing A into a training and a test set, called `A_train` and `A_test`, then solve for the optimal decoders. `y_train` and `y_test` correspond to the desired output rates for the test and training sets.

```
########################################
# train population
# The notation A’ means “A transpose”
# y = A'd
# Ay = AA'd
# inv(AA')Ay = d
# inv(AA' + lam*I)Ay = d
#
# try different lambdas, pick the best one
# increase if necessary to keep weights < 1

def rmse(x, y):
     return np.sqrt(np.mean((x - y)**2))

# sweep lambda
decoders = []
rmses = []
for lam in lams:
     d = np.dot(np.dot(np.linalg.inv(np.dot(A_train, A_train.T) +
lam * np.eye(N)), A_train), ytrain)
     yhat = np.dot(A_test.T, d)
     cv_rmse = rmse(ytest, yhat)

     decoders.append(d)
     rmses.append(cv_rmse)

# pick the best decode for all lambdas
```

```
best_idx = np.argmin(rmses)
print("best lambda was", lams[best_idx])
print("  with RMSE", np.min(rmses))

best_d = decoders[best_idx]

impl_idx = best_idx
impl_d = decoders[impl_idx]
```

Once we've used this data to determine the proper decoders, we need to call **HAL.remap()** to remap the new decoders to the chip, then use RunControl again to run another sweep.

```
########################################
# set decode weights, call HAL.remap_weights()

# all we have to do is change the decoders
decoder_conn.reassign_weights(impl_d.T)

# then call remap
print("remapping weights")
HAL.remap_weights()

# times are in nanoseconds (1e9 ns = 1 sec)
FUDGE = 2
curr_time = HAL.get_time()
times = np.arange(0, total_training_points) * training_hold_time *
1e9 + curr_time + FUDGE * 1e9
times_w_end = np.hstack((times, times[-1] + training_hold_time *
1e9))
vals = np.array(stim_rates).T
input_vals = {i1 : (times, vals)}

rc = RunControl(HAL, net)

# collect the binned bucket threshold events
outputs_and_bin_times, _ = rc.run_input_sweep(input_vals,
get_raw_spikes=False, end_time=times_w_end[-1], rel_time=False)
events, event_bin_times = outputs_and_bin_times
yhat = data_utils.bins_to_rates(events[o1], event_bin_times,
times_w_end, init_discard_frac=.2)
yhat = yhat.flatten()
```

Finally, we plot our output. The x-axis is the input stimulation rate for the network, while the outputs are the output rates of the system after the decode. This allows us to confirm that we successfully programmed decoders that actually decode the function we're trying to decode.

```
    plot_x = stim_rates_1d
    ytgt = (np.array(y_rates).T).flatten()
    test_rmse = rmse(ytgt, yhat)
    plt.figure()
    plt.plot(plot_x, ytgt, color='b')
    plt.plot(plot_x, yhat, color='r')
    plt.title("Accumulator Decode\nRMSE: " + str(test_rmse))
    plt.legend(["target", "FPGA SF outputs"])
    plt.xlabel("input rate")
    plt.ylabel("output rate")
    plt.savefig("hal_acc_decode.pdf")
    print("rmse for accumulator decode:", test_rmse)
    assert(test_rmse <= 1.5 * impl_rmse) # we want the training
RMSE to be similar to the test RMSE
```