```
#-------------------------------------------------------------------------------
# Imports
#-------------------------------------------------------------------------------

import numpy as np

from pycog import Model, RNN, tasktools
```

Imports “numpy” library with nickname “np”

And from pycog module Model, RNN and tasktools (essentially it is importing Model and RNN class from the model.py and rnn.py inside pycog P.S. refer here to know more about python classes)

```
#-------------------------------------------------------------------------------
# Network structure
#-------------------------------------------------------------------------------

Nin  = 2
N    = 100
Nout = 2

# E/I
ei, EXC, INH = tasktools.generate_ei(N)
print("Your ei : {}".format(ei))
print("Your EXC : {}".format(EXC))
print("Your INH : {}".format(INH))
# Output connectivity: read out from excitatory units only
Cout = np.zeros((Nout, N))
Cout[:,EXC] = 1
```

The above defines the some structure of network like input units, hidden units and output units along with excitatory/inhibitory units value and a list “ei” that contains 1 and –1 values in ratio of excitatory to inhibitory units I.e 80/20 (80 : +1, 20: -1)

It does so by calling tasktools.py file’s “generate_ei” function

```
def generate_ei(N, pE=0.8):
    """
    E/I signature.

    Parameters
    ----------

    N : int
        Number of recurrent units.

    pE : float, optional
         Fraction of units that are excitatory. Default is the usual value for
cortex.

    """
    assert 0 <= pE <= 1

    Nexc = int(pE*N)
    Ninh = N - Nexc

    idx = range(N)
    EXC = idx[:Nexc]
    INH = idx[Nexc:]

    ei       = np.ones(N, dtype=int)
    ei[INH] *= -1

    return ei, EXC, INH
```

This function takes number of hidden units and ratio of excitatory to inhibitory unit as argument, by default the ratio is set to 0.8(Default is the usual value for cortex.). hence in case pE is not provided it will automatically assume it as 0.8

Next the assert statement validates the range of pE, so just in case the provided value exceeds the range it will throw an error

Then it assigns the number of excitatory and inhibitory units' number in form of variable by multiplying the ratio with number of hidden units

The idx variable is generated with range function that takes N as argument and creates a generator object that when iterated will produce values from 0 to N-1

After this the idx range is split for excitatory and inhibitory units I.e. range 0 to 80 is excitatory and 80 till 100 is inhibitory unit range.

Now by using numpy's ones function an array is created of size "N" with only +1 values and datatype integer. Then lastly this array is modified for replacing last 80 to 100 +ones to negative ones hence denoting injhibitory part

Finally this "ei", EXC, INH is returned.

The values returned are as follows:

```
Your ei : [ 1  1  1  1  1  1  1  1  1  1  1  1  1  1  1  1  1  1  1  1  1  1  1  1
  1  1  1  1  1  1  1  1  1  1  1  1  1  1  1  1  1  1  1  1  1  1  1  1
  1  1  1  1  1  1  1  1  1  1  1  1  1  1  1  1  1  1  1  1  1  1  1  1
  1  1  1  1  1  1  1  1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1
 -1 -1 -1 -1]
Your EXC : range(0, 80)
Your INH : range(80, 100)
```

(ei has 100 values:- 80 : +1, 20 : -1)

Lastly the "Cout" variable represent output connectivity of model. np.zeros function creates a matrix of only zero values with size as provided in argument. np.zeros(*row_size* , *column_size*)

In our case *row_size* is Nout (values : 2) and column_size is N (value : 100)

Then this Cout specifically overwritten to keep only columns in range of excitatory units to be 1 else will remain zero(done so by list slicing list is sliced as : *list_name(row_start : row_end , column_start) : column_end)* a single ':' with no start-stop means to take from 0 to last index)

Such configuration allows only to take info only from excitatory units for output

Cout:

```
array([[1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1.,
        1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1.,
        1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1.,
        1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1.,
        1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1.,
        0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.,
        0., 0., 0., 0.],
       [1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1.,
        1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1.,
        1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1.,
        1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1.,
        1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1.,
        0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.,
        0., 0., 0., 0.]])
```

```
#-------------------------------------------------------------------------------
# Task structure
#-------------------------------------------------------------------------------
cohs        = [1, 2, 4, 8, 16]
left_rights = [1, -1]
nconditions = len(cohs)*len(left_rights)
pcatch      = 1/(nconditions + 1)

SCALE = 3.2
def scale(coh):
    return (1 + SCALE*coh/100)/2
```

this section defines task structure of the model

the ‘cohs’ variable is a list that contain 5 different values of coherence

left_rights variable is a list that store 2 values +1 and -1 conventionally rep[resenting left and right in numerical form

the nconditions variable is multiplication of length of cohs list and left_right list i.e 5*2 = 10, defining number of conditions or number of possible pairs of cohs and left_right values

the pcatch variable stores probability of catch trial happening that is given by 1/(nconditions+1) putting nconditions=10 we get pcatch=1/11 or nearly 9.09% chance.

Scale is a integer variable with value 3.2

Scale function is defined that scales and returns the value of coherence according to a formula:

1 + Scale*coh/100)/2

```
def generate_trial(rng, dt, params):
```

Now comes the most essential and prime part of the S1_code file, the “generate_trial” function that is a necessity for every model file to be trained by pycog to have without it the pycog can’t get parameters, inputs and outputs information required for the model training and testing. (all files in examples/models folder have generate trial function)

The function as of now is not supplied with anything it is just **defined** here it’s calculations and logic will only trigger when it is called in training

The function takes 3 arguments namely:-

rng : numpy.random.RandomState

A numpy object that can generate of random numbers from variety of probability distribution when needed (refer here for more info about randomstate).

dt : a float value

Representing integration time step.

Params : a dictionary data structure variable (refer here to know about python dictionary)

contains initial parameters for generating trial

initially their values are

```
Your params: {'callback_results': None, 'target_output': True, 'name':'gradient'}
Your dt: 20.0
Your rng is :RandomState(MT19937)
```

Params:

callback_results : any, optional argument

contains Information used to adapt training.

target_output : Boolean variable

specifies if there is target output for this model or not

has values: True or False

name: String variable

Name of the dataset, which can be used by `task` in dataset file (a python function), e.g., to distinguish between gradient and validation datasets.

dt:

float variable

contains gradient minibatch size

rng:

Container for the Mersenne Twister pseudo-random number generator.
Target use being generating random numbers.

For more info refer here

```
#------------------------------------------------------------------------
# Select task condition
#------------------------------------------------------------------------

if params.get('catch', rng.rand() < pcatch):
    catch_trial = True
else:
    catch_trial = False
    coh         = params.get('coh',        rng.choice(cohs))
    left_right  = params.get('left_right', rng.choice(left_rights))
```

here comes selecting task conditions deciding whether trial will be catch trial or not and add constraints accordingly

params is a dictionary object and dictionary objects have .get() function that when called with an argument key provided returns that keys values present in the dictionary(in our case “catch” is the key) and if a second argument is also provided returns that second argument in case the forementioned key is not present, for our case it is a boolean statement given as the second argument where “rng.rand()” will generate a random number in between 0 to 1 and pcatch (value : 0.09090), the statement is that random number is lesser than pcatch if it holds then answer will be true and second argument will be ‘True’ else ‘False’ so as value of pcatch is 0.09090 the chances for the condition to hold true will be 9.09% hence 9.09% chance for a catch trial to be assigned “True” value or in other words 9.09% chance for catch trial

now if the condition gets false the if statement will not get called and moves on to else part here catch_trial will get assigned “False”, for coh and left_rights by same logic .get() function, if key ‘coh’ and left_right are not found it will give ‘rng.choice(cohs)’ and ‘rng.choice(left_rights)’ instead. The .choice() will crandomly choose any one of the value from given list/array provided as argument and return it

hence coh and left_right will both get assigned randomly a value from cohs and left_rights list created before.

```
#--------------------------------------------------------------------------
# Epochs
#--------------------------------------------------------------------------

if catch_trial:
    epochs = {'T': 2000}
else:
    fixation = 100
    stimulus = 800
    decision = 300
    T        = fixation + stimulus + decision

    epochs = {
        'fixation': (0, fixation),
        'stimulus': (fixation, fixation + stimulus),
        'decision': (fixation + stimulus, T)
        }
    epochs['T'] = T
```

If catch_trial variable is 'True' if condition will trigger which will create a 'epoch' dictionary containing number of trial 'T' as key and value 2000

Otherwise, 3 variable 'fixation', 'stimulus' and 'decision' will get created with values 100,800 and 300 value respectively

Then 'T' will be assigned the sum the above 3 and lastlty epoch dictionary will be created with 'fixation', 'stimulus' and 'decision' as keys and a tuple as value to each signifying their span of interval according to above code(a tuple is pretty similar to lists except the difference in their mutability and handling and that is irrelevant here) finally they add "T" key with value variable 'T' in dictionary(which I think they could have also done whilst they created the dictionary though it doesn't matter anyways it's done)

```
    #---------------------------------------------------------------------------
    # Trial info
    #---------------------------------------------------------------------------

    t, e  = tasktools.get_epochs_idx(dt, epochs) # Time, task epochs
    trial = {'t': t, 'epochs': epochs}           # Trial
    # print("Your t,e is {}".format((t, e)))
    if catch_trial:
        trial['info'] = {}
    else:
        # Correct choice
        if left_right > 0:
            choice = 0
        else:
            choice = 1

        # Trial info
        trial['info'] = {'coh': coh, 'left_right': left_right, 'choice': choice}
```

here comes trial info section which comprises of variables definition relevant to a particular trial

variable ‘t’ and ‘e’ are created by using tasktools.py files “get_epochs_idx()” function that takes time step and epochs dict created before as argument

```
#-------------------------------------------------------------------------------
# Functions for defining task epochs
#-------------------------------------------------------------------------------

def get_idx(t, interval):
    start, end = interval

    return list(np.where((start < t) & (t <= end))[0])

def get_epochs_idx(dt, epochs):
    t = np.linspace(dt, epochs['T'], int(epochs['T']/dt))
    # assert t[1] - t[0] == dt, "[ tasktools.get_epochs_idx ] dt doesn't fit into
T."

    return t, {k: get_idx(t, v) for k, v in epochs.items() if k != 'T'}
```

here linspace function of numpy library is used to to create an array of numbers then stored in ‘t’ variable

linspace take 3 arguments: start, stop, total count of numbers to generate

for our case we have start as dt(i.e 20.0), epochs[‘T’] as stop (i.e either 2000 or 1200 acc to catch or not) and total numbers as **integer value** of ‘epochs[‘T’]/dt’ (i.e. 2000/20=100 or 1200/20=60)

the next commented line checks if difference between 2 units of ‘t’ is dt or not. If not then it throws an assertion error.

Afterwards it returns ‘t’ and an dictionary which has all keys present in ‘epochs’(the one supplied as argument) as its own keys(except for ‘T’) and their values as returned value from get_idx() function of very same tasktools file

The get_idx takes ‘t’(created above) and values of each key present in epochs dictionary (one by one) as argument

The get_idx function separates start and stop from interval which is a tuple with 2 values (epochs dictionary’s values which are tuple can refer to some parts above to see the epochs dict)

Finally it returns index of every value in ‘t’ that is greater than start but less than stop

This is done using numpy library’s where function which implements this condition and returns index(can also return values) in form of, array of numbers and datatype all in an array from which array is extracted by slicing (taking only 0^{th} index element i.e. where array is present)

For more info about where function refer here

This list is returned for every key-value pair in epochs(except ‘T’ key).

The return of get_epochs_idx is as follows:

```
Your t,e is (
array([  20.,   40.,   60.,   80.,  100.,  120.,  140.,  160.,  180.,
        200.,  220.,  240.,  260.,  280.,  300.,  320.,  340.,  360.,
        380.,  400.,  420.,  440.,  460.,  480.,  500.,  520.,  540.,
        560.,  580.,  600.,  620.,  640.,  660.,  680.,  700.,  720.,
        740.,  760.,  780.,  800.,  820.,  840.,  860.,  880.,  900.,
        920.,  940.,  960.,  980., 1000., 1020., 1040., 1060., 1080.,
       1100., 1120., 1140., 1160., 1180., 1200.]),
         {'fixation': [0, 1, 2, 3, 4],
         'stimulus': [5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20,
21, 22, 23, 24, 25, 26, 27, 28, 29, 30, 31, 32, 33, 34, 35, 36, 37, 38, 39, 40,
41, 42, 43, 44],
         'decision': [45, 46, 47, 48, 49, 50, 51, 52, 53, 54, 55, 56, 57, 58,
59]}
)
```

Moving on this t, e captured in separate variables are then stored in a new dict “trial” implying dictionary storing trial information, this dict is also will be the one to returned from generate_trial function

Next catch_trial instance is checked if yes: then key ‘info’ is added with empty dictionary as value i.e. trial info is left empty for catch trial

Else:

choice is assigned according to left_right taken in an instance of trial.

Key ‘info’ is added with value in form of dictionary with the ‘coh’, ‘left_right’ and ‘choice’ as keys and values.

Trial dict right now:

```
Your Trial is {
't': array([  20.,   40.,   60.,   80.,  100.,  120.,  140.,  160.,  180.,
        200.,  220.,  240.,  260.,  280.,  300.,  320.,  340.,  360.,
        380.,  400.,  420.,  440.,  460.,  480.,  500.,  520.,  540.,
        560.,  580.,  600.,  620.,  640.,  660.,  680.,  700.,  720.,
        740.,  760.,  780.,  800.,  820.,  840.,  860.,  880.,  900.,
        920.,  940.,  960.,  980., 1000., 1020., 1040., 1060., 1080.,
       1100., 1120., 1140., 1160., 1180., 1200.]),

'epochs': {
      'fixation': (0, 100),
      'stimulus': (100, 900),
      'decision': (900, 1200),
      'T': 1200
      },

 'info':
 {'coh': 2, 'left_right': -1, 'choice': 1}}
```

```
#--------------------------------------------------------------------------
    # Inputs
    #--------------------------------------------------------------------------

    X = np.zeros((len(t), Nin))
    if not catch_trial:
        X[e['stimulus'],choice]   = scale(+coh)
        X[e['stimulus'],1-choice] = scale(-coh)
    trial['inputs'] = X
```

this section defines input for trial instance

starting with creating 'X' variable with zeros array of size: rows- length of 't' and columns- Nin

value: 60 or 100 value:2

Now if trial isn't a catch_trial i.e. a stimulus trial

then stimulus parts of X will be overwritten with scaled coherence according to 'choice' of that trial

according to which is prominent choice the coherence will be scaled and assigned, for the favorable choice positive coherence is used for scaling and for other choice, negative coherence

lastly regardless of catch trial or not input key is attached to trial dictionary with 'X' array as value

For catch trial 'X' will comprise of only zeros and for normal trial it will be different than zero according to coherence choosen for that trial, will differ.

Example X (for coh=2 and left_right = -1, choice 1):

```
Your X is :
 array([[0.   , 0.   ],
       [0.   , 0.   ],
       [0.   , 0.   ],
       [0.   , 0.   ],
       [0.   , 0.   ],
       [0.468, 0.532],
       [0.468, 0.532],
       [0.468, 0.532],
       [0.468, 0.532],
       [0.468, 0.532],
       [0.468, 0.532],
       [0.468, 0.532],
       [0.468, 0.532],
       [0.468, 0.532],
       [0.468, 0.532],
```

```
       [0.468, 0.532],
       [0.468, 0.532],
       [0.468, 0.532],
       [0.468, 0.532],
       [0.468, 0.532],
       [0.468, 0.532],
       [0.468, 0.532],
       [0.468, 0.532],
       [0.468, 0.532],
       [0.468, 0.532],
       [0.468, 0.532],
       [0.468, 0.532],
       [0.468, 0.532],
       [0.468, 0.532],
       [0.468, 0.532],
       [0.468, 0.532],
       [0.468, 0.532],
       [0.468, 0.532],
       [0.468, 0.532],
       [0.468, 0.532],
       [0.468, 0.532],
       [0.468, 0.532],
       [0.468, 0.532],
       [0.468, 0.532],
       [0.468, 0.532],
       [0.468, 0.532],
       [0.468, 0.532],
       [0.468, 0.532],
       [0.468, 0.532],
       [0.468, 0.532],
       [0.468, 0.532],
       [0.   , 0.   ],
       [0.   , 0.   ],
       [0.   , 0.   ],
       [0.   , 0.   ],
       [0.   , 0.   ],
       [0.   , 0.   ],
       [0.   , 0.   ],
       [0.   , 0.   ],
       [0.   , 0.   ],
       [0.   , 0.   ],
       [0.   , 0.   ],
       [0.   , 0.   ],
       [0.   , 0.   ],
       [0.   , 0.   ],
```

```
       [0.   , 0.   ]])
```

```
    #---------------------------------------------------------------------------
    # Target output
    #---------------------------------------------------------------------------

    if params.get('target_output', False):
        Y = np.zeros((len(t), Nout)) # Output
        M = np.zeros_like(Y)         # Mask

        if catch_trial:
            Y[:] = 0.2
            M[:] = 1
        else:
            # Fixation
            Y[e['fixation'],:] = 0.2

            # Decision
            Y[e['decision'],choice]   = 1
            Y[e['decision'],1-choice] = 0.2

            # Only care about fixation and decision periods
            M[e['fixation']+e['decision'],:] = 1

        # Outputs and mask
        trial['outputs'] = Y
        trial['mask']    = M
```

Section for output definitions:

First checks if params has ‘target_output’ or not if not then False if yes then value of ‘target_output’ which if either True or False depending on what is supplied in params for all cases in trial of S1_code it is ‘True’

So assuming it’s True:

Y(output) and M (Mask) zero arrays are created of size: rows length of t and column : Nout

Values: 60 or 100 , 2

If catch trial output has al values set to 0.2 and Mask’s to 1

Otherwise, fixation period output is set 0.2 and decision period: favorable choice 1 else set 0.2

Mask’s decision and fixation range is assigned 1

Finally regardless of catch trial or not trial dict is filled with output and mask key-value pair

```
    return trial
```

Lastly this trial dictionary is returned to wherever it was called

Final trial dict's example:

```
Your Trial is {
't': array([  20.,   40.,   60.,   80.,  100.,  120.,  140.,  160.,  180.,
        200.,  220.,  240.,  260.,  280.,  300.,  320.,  340.,  360.,
        380.,  400.,  420.,  440.,  460.,  480.,  500.,  520.,  540.,
        560.,  580.,  600.,  620.,  640.,  660.,  680.,  700.,  720.,
        740.,  760.,  780.,  800.,  820.,  840.,  860.,  880.,  900.,
        920.,  940.,  960.,  980., 1000., 1020., 1040., 1060., 1080.,
       1100., 1120., 1140., 1160., 1180., 1200.]),

'epochs': {
      'fixation': (0, 100),
      'stimulus': (100, 900),
      'decision': (900, 1200),
      'T': 1200
      },

 'info':
 {'coh': 2, 'left_right': -1, 'choice': 1},
 'inputs':
 array([[0.   , 0.   ],
       [0.   , 0.   ],
       [0.   , 0.   ],
       [0.   , 0.   ],
       [0.   , 0.   ],
       [0.468, 0.532],
       [0.468, 0.532],
       [0.468, 0.532],
       [0.468, 0.532],
       [0.468, 0.532],
       [0.468, 0.532],
       [0.468, 0.532],
       [0.468, 0.532],
       [0.468, 0.532],
       [0.468, 0.532],
       [0.468, 0.532],
       [0.468, 0.532],
       [0.468, 0.532],
       [0.468, 0.532],
```

```
       [0.468, 0.532],
       [0.468, 0.532],
       [0.468, 0.532],
       [0.468, 0.532],
       [0.468, 0.532],
       [0.468, 0.532],
       [0.468, 0.532],
       [0.468, 0.532],
       [0.468, 0.532],
       [0.468, 0.532],
       [0.468, 0.532],
       [0.468, 0.532],
       [0.468, 0.532],
       [0.468, 0.532],
       [0.468, 0.532],
       [0.468, 0.532],
       [0.468, 0.532],
       [0.468, 0.532],
       [0.468, 0.532],
       [0.468, 0.532],
       [0.468, 0.532],
       [0.468, 0.532],
       [0.468, 0.532],
       [0.468, 0.532],
       [0.468, 0.532],
       [0.468, 0.532],
       [0.468, 0.532],
       [0.468, 0.532],
       [0.   , 0.   ],
       [0.   , 0.   ],
       [0.   , 0.   ],
       [0.   , 0.   ],
       [0.   , 0.   ],
       [0.   , 0.   ],
       [0.   , 0.   ],
       [0.   , 0.   ],
       [0.   , 0.   ],
       [0.   , 0.   ],
       [0.   , 0.   ],
       [0.   , 0.   ],
       [0.   , 0.   ],
       [0.   , 0.   ],
       [0.   , 0.   ]]),

'outputs':
```

```
array([[0.2, 0.2],
       [0.2, 0.2],
       [0.2, 0.2],
       [0.2, 0.2],
       [0.2, 0.2],
       [0. , 0. ],
       [0. , 0. ],
       [0. , 0. ],
       [0. , 0. ],
       [0. , 0. ],
       [0. , 0. ],
       [0. , 0. ],
       [0. , 0. ],
       [0. , 0. ],
       [0. , 0. ],
       [0. , 0. ],
       [0. , 0. ],
       [0. , 0. ],
       [0. , 0. ],
       [0. , 0. ],
       [0. , 0. ],
       [0. , 0. ],
       [0. , 0. ],
       [0. , 0. ],
       [0. , 0. ],
       [0. , 0. ],
       [0. , 0. ],
       [0. , 0. ],
       [0. , 0. ],
       [0. , 0. ],
       [0. , 0. ],
       [0. , 0. ],
       [0. , 0. ],
       [0. , 0. ],
       [0. , 0. ],
       [0. , 0. ],
       [0. , 0. ],
       [0. , 0. ],
       [0. , 0. ],
       [0. , 0. ],
       [0. , 0. ],
       [0. , 0. ],
       [0. , 0. ],
       [0. , 0. ],
       [0. , 0. ],
```

```
       [0.2, 1. ],
       [0.2, 1. ],
       [0.2, 1. ],
       [0.2, 1. ],
       [0.2, 1. ],
       [0.2, 1. ],
       [0.2, 1. ],
       [0.2, 1. ],
       [0.2, 1. ],
       [0.2, 1. ],
       [0.2, 1. ],
       [0.2, 1. ],
       [0.2, 1. ],
       [0.2, 1. ],
       [0.2, 1. ]]),

'mask':
array([[1., 1.],
       [1., 1.],
       [1., 1.],
       [1., 1.],
       [1., 1.],
       [0., 0.],
       [0., 0.],
       [0., 0.],
       [0., 0.],
       [0., 0.],
       [0., 0.],
       [0., 0.],
       [0., 0.],
       [0., 0.],
       [0., 0.],
       [0., 0.],
       [0., 0.],
       [0., 0.],
       [0., 0.],
       [0., 0.],
       [0., 0.],
       [0., 0.],
       [0., 0.],
       [0., 0.],
       [0., 0.],
       [0., 0.],
       [0., 0.],
       [0., 0.],
```

```
       [0., 0.],
       [0., 0.],
       [0., 0.],
       [0., 0.],
       [0., 0.],
       [0., 0.],
       [0., 0.],
       [0., 0.],
       [0., 0.],
       [0., 0.],
       [0., 0.],
       [0., 0.],
       [0., 0.],
       [0., 0.],
       [0., 0.],
       [0., 0.],
       [0., 0.],
       [0., 0.],
       [1., 1.],
       [1., 1.],
       [1., 1.],
       [1., 1.],
       [1., 1.],
       [1., 1.],
       [1., 1.],
       [1., 1.],
       [1., 1.],
       [1., 1.],
       [1., 1.],
       [1., 1.],
       [1., 1.],
       [1., 1.],
       [1., 1.]])}
```

```
# Performance measure: two-alternative forced choice
performance = tasktools.performance_2afc
```

defines performance function to use to evaluate model, here they used tasktools performance_2afc function which is based on two-alternative forced choice performance measure

refer here for more info about this measure

```
# Terminate training when psychometric performance exceeds 85%
def terminate(performance_history):
    return np.mean(performance_history[-5:]) > 85

# Validation dataset size
n_validation = 100*(nconditions + 1)
```

termination function mapping the condition for stopping training, it does so by checking the mean average of last 5 performances being greater than 85, if yes returns 'True' else returns 'False'.

As can be seen it takes performance_history as argument (to be provided in by some other file's function whilst training)

Last 2 lines define validation datasize as 100 times of total number of condition/cases +1

100*(10+1)=1100

This marks the end of modelfile specific things that define this model's training information, criteria , inputs, outputs etc.

The afterwards code is a driver code that kickstarts training using model.py Model function

After training it also runs the network

Note: An almost alternative of this part is using 'python do.py *{model_file}* train' which will just train the model but not run it

```
if __name__ == '__main__':
    # Train model
    model = Model(Nin=Nin, N=N, Nout=Nout, ei=ei, Cout=Cout,
                  generate_trial=generate_trial,
                  performance=performance, terminate=terminate,
                  n_validation=n_validation)
    model.train('savefile.pkl')

    # Run the trained network with 16*3.2% = 51.2% coherence for choice 1
    rnn        = RNN('savefile.pkl', {'dt': 0.5})
    trial_func = generate_trial
    trial_args = {'name': 'test', 'catch': False, 'coh': 16, 'left_right': 1}
    info       = rnn.run(inputs=(trial_func, trial_args))
    rnn.plot
    print(info)
```

the if condition defined here is always true as long as this file is run directly, in case if some other module access this file's content then the condition will not hold true as then __name__ variable will not be string '__main__'. For more info about __name__ special variable refer here.

```
    model = Model(Nin=Nin, N=N, Nout=Nout, ei=ei, Cout=Cout,
                  generate_trial=generate_trial,
                  performance=performance, terminate=terminate,
                  n_validation=n_validation)
```

Model class from model.py in pycog is **initialized** here as model variable with arguments: Number of inputs , Number of hidden layers, excitatory-inhibitory units defining array, output connectivity matrix, generate_trial function, performance evaluating function, termination function and validation data size

By initializing we are creating an **instance** of the Model class under a name 'model' with as specified arguments (refer here to know about instance of class)

Moving over to model file and let's see what happens with these arguments further

```
class Model:
    """
    Provide a simpler interface to users, and check for obvious mistakes.

    """
```

This is the class that is called from model file

```
 def __init__(self, modelfile=None, **kwargs):
        """
        If `modelfile` is provided infer everything from the file, otherwise the
        user is responsible for providing the necessary parameters through
`kwargs`.

        Parameters
        ----------

        modelfile: str
                   A Python script containing model parameters.

        kwargs : dict

          Should contain

          generate_trial : function
                           Return a dictionary containing trial information.
                           This function should take as its arguments
```

```
                        rng    : numpy.random.RandomState
                        dt     : float
                        params : dict

        """
```

Here __init__ function is defined that has a property to get automatically get triggered whenever Model class is initialized i.e. a class object is created, just like in our case we have initialized it with 'model' variable in S1_code.py file

For more info about __init__ function refer here

For every method inside every class they must first take 'self' as input it is requirement for every class method. This 'self' can be very resourceful as we can make instance specific variables in it, store and also edit that data between different functions of this same class. (It can do more than what is explained here refer here)

For example:

Let's say you have a class ClassA which contains a method methodA defined as:

```
def methodA(self, arg1, arg2):
    # do something
```

and ObjectA is an instance of this class.

Now when

```
ObjectA.methodA(arg1, arg2)
```

is called, python internally converts it for you as:

```
ClassA.methodA(ObjectA, arg1, arg2)
```

The self variable refers to the object itself.

(Example taken from one of Stackoverflow's forum check here)

Modelfile argument here is intended to hold path of modelfile to be provided with initialization of model class, if not provided it will take 'None' as value by default. This method of providing model file is used by do.py on the other hand what S1_code.py file uses for training is **kwargs argument, using this argument it takes all the items provided to Model() and convert them to dictionary format. Hence all argument provided to Model() in S1_code coverts to dictionary with items on left hand side of equal sign as key and right hand side ones as their value. (For knowing about **kwargs refer here)

**kwargs and modelfile variable values for S1_code.py

```
Your kwargs:  {'Nin': 2, 'N': 100, 'Nout': 2, 'ei': array([ 1,  1,  1,  1,  1,
1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,
       1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,
       1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,
       1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,
       1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1, -1, -1, -1, -1, -1,
      -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1]), 'Cout':
array([[1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1.,
        1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1.,
        1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1.,
        1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1.,
        1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1.,
        0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.,
        0., 0., 0., 0.],
       [1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1.,
        1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1.,
        1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1.,
        1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1.,
        1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1.,
        0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.,
        0., 0., 0., 0.]]), 'generate_trial': <function generate_trial at
0x0000024B15725670>, 'performance': <function performance_2afc at
0x0000024B158479D0>, 'terminate': <function terminate at 0x0000024B1584F670>,
'n_validation': 1100}
Your modelfile:  None
```

```
        if modelfile is not None:
            try:
                spec2 = importlib.util.spec_from_file_location('model',modelfile)
                self.m = importlib.util.module_from_spec(spec2)
                spec2.loader.exec_module(self.m)

            except IOError:
                print("[ {}.Model ] Couldn't load model file {}".format(THIS,
modelfile))
                sys.exit(1)
        else:
            self.m = Struct(**kwargs)
```

the above code checks if modelfile is provided if it has been provided such, it will *try* to load that file using importlib library what it does basically is load that module's('module' refering to python file) all function under the hood of 'self.m' i.e Now using self.m we can call any function/variable present in S1_code

What try and except does is it tries to run code in 'try' if it works then it moves on with it otherwise if there's an error encountered in the execution it will move on to executing code under 'except' block, this is an useful method to code cases where we are unsure about correct execution of some code that may encounter some error depending on it's state of execution so we can also handle cases where it encounters some error. For more info about 'try' and ''except' refer here

if the importing of file results in an error for example if the specified module file doesn't exist or name spelled might be wrong then it will run exception statement and print where {} are space holders and will be replaced there by arguments in .format(), 'THIS' is defined at start of model file and has string 'pycog.model' and using 'sys.exit' it will close the further execution.

While if modelfile is None that probably means **kwargs are provided so it will create self.m an instance of class Struct with **kwargs as arguments supplied to it (for Struct's init function)

```
class Struct:
    """
    Treat a dictionary like a module.

    """
    def __init__(self, **entries):
        self.__dict__.update(entries)
```

This function as said converts the dictionary in a module

As soon as Struct object is created __init__ gets triggered with usual class default ‘self’ argument and ‘**entries’ argument, as we gave **kwargs as argument to Struct the **entries will be **kwargs just under different local name

Now the thing is self.__dict__ contains all functions and variable of its class in form of dict, so by updating this dict we are essentially creating new variables and function from the dict we supplied to this new class. Hence now it possess all functions and variables present in **kwargs.

So, now we have our self.m ready, regardless of how it was created (either by modelfile or **kwargs) it will contain functions and variables from modelfile we are using for training(For our case it is ‘S1_code.py’)

```
Your entries {'Nin': 2, 'N': 100, 'Nout': 2,
'ei': array([ 1,  1,  1,  1,  1,  1, 1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,
       1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,
       1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,
       1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,
       1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1, -1, -1, -1, -1, -1,
      -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1]),
'Cout': array([[1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1.,
       1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1.,
       1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1.,
       1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1.,
       1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1.,
       0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.,
       0., 0., 0., 0.],
      [1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1.,
       1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1.,
       1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1.,
       1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1.,
       1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1.,
       0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.,
       0., 0., 0., 0.]]),
'generate_trial': <function generate_trial at 0x0000016E6F4F5670>,
'performance': <function performance_2afc at 0x0000016E6F6179D0>,
'terminate': <function terminate at 0x0000016E6F6629D0>,
'n_validation': 1100}

Prev self dict:  {}

New self dict:  {'Nin': 2, 'N': 100, 'Nout': 2, 'ei': array([ 1,  1,  1,  1,  1,
1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,
       1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,
       1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,
       1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,
       1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1, -1, -1, -1, -1, -1,
```

```
      -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1]), 'Cout':
array([[1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1.,
        1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1.,
        1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1.,
        1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1.,
        1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1.,
        0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.,
        0., 0., 0., 0.],
       [1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1.,
        1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1.,
        1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1.,
        1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1.,
        1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1.,
        0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.,
        0., 0., 0., 0.]]), 'generate_trial': <function generate_trial at
0x0000016E6F4F5670>, 'performance': <function performance_2afc at
0x0000016E6F6179D0>, 'terminate': <function terminate at 0x0000016E6F6629D0>,
'n_validation': 1100}
```

In case do.py was used for training then:

```
Your modelfile:  E:\pycog-master\examples\models\S1_code.py
Your kwargs:  {}
```

```
        #-------------------------------------------------------------------------
        # Perform model check
        #-------------------------------------------------------------------------

        try:
            f = self.m.generate_trial
            args = inspect.getfullargspec(f).args
            print("Your f is: ", f)
            print("Your args is: ", args)

        except AttributeError:
            print("[ {}.Model ] You need to define a function that returns
trials.".format(THIS))
            sys.exit(1)
```

Now the code performs model check, by far we have a class object that has all required function and variables for the model to be trained, so we will do a check if it possesses everything essential for training

Under the try block:

it will create ‘f’ named variable that will hold generate trial function from self.m (it’s not like self.m has now lost generate trial function it’s just generate trial function has now a shorter alias name for calling/using wherever needed)

Next they get all arguments that are present in generate trial function using inspect. getfullargspec () function and store them as ‘args’ variable, it is retrieved in form of list.

What getfullargspec does is, it returns a list like object containing all arguments and specification type needed by that function. by providing ‘.args’ at the end we are specifically demanding only arguments present in that list. For more info about getfullargspec() refer here

```
Your f is:  <function generate_trial at 0x000001E191F34040>
Your args is:  ['rng', 'dt', 'params']
```

Under except block:

In case there’s an attribute error i.e. there’s no function known as generate trial or some similar error then this except code block will execute and give user an error that they need to define a trial function first.

Sys.exit() closes further execution

```
        # generate_trial : usage
        if args != ['rng', 'dt', 'params']:
            print(("[ {}.Model ] Warning: Function generate_trial doesn't have
the"
                   " expected list of argument names. It is OK if only the names
are"
                   " different.").format(THIS))
```

Checks if the args list is same as desired i.e. required arguments are present in the generate trial function, if not then give out warning as seen in above code just in case names of argument are different.

```
        # var_in : size
        if (hasattr(self.m, 'var_in') and isinstance(self.m.var_in, np.ndarray)
                and self.m.var_in.ndim == 1):
            if len(self.m.var_in) != self.m.Nin:
                print(
                    "[ {}.Model ] The length of var_in doesn't match
Nin.".format(THIS))
                sys.exit(1)
```

Next it checks variance in input's size

'var_in' represent variance of input, the code checks, if this attribute is provided, is an numpy n dimensional array and if it's dimension is 1 if these 3 hold true then it checks if variance array size is equal to number inputs or not, if not then print length doesn't match and close execution

Hasattr() function checks if provided object has given attribute or not; returns True or False

Isinstance() function checks if given object is an instance of given class or not, here it checks if self.m.var_in is an ndarray or not i.e. part of class ndarray of numpy; returns True or False

Both are inbuilt functions and for our case of S1_code none of the condition holds true

```
        # if terminate is given, performance should also be given
        if hasattr(self.m, 'terminate') and not hasattr(self.m, 'performance'):
            print(("[ {}.Model ] Warning: Termination criterion is provided, "
                   " but the performance measure is not defined").format(THIS))
```

This code maps the condition where terminate function is given but performance function is absent ; in this case it will print out a warning to user. The logic of hasattr is same as previous.

This marks the end for __init__ execution that was triggered in S1_code. Further execution will be triggered back from S1_code

---

Back in S1_code

```
    model.train('savefile.pkl')
```

the model object created just before this line is used to call its train method i.e., train function exclusive to its class.

the string 'savefile.pkl' is provided as an argument; it is the file name for the created mode's data .pkl is an extension for files created from pickle module in python that will be created up ahead in trainer file

Pickle: a Python module that enabless objects to be serialized to files on disk and deserialized back into the program at runtime. It contains a byte stream that represents the objects.

Moving over to train in model.py

```
    def train(self, savefile, seed=None, compiledir=None, recover=True, gpus=0):
        """
        Train the network.

        Parameters
        ----------

        savefile : str
        seed : int, optional
        compiledir : str, optional
        recover : bool, optional
        gpus : int, optional

        """
```

It takes mandatory arguments self and savefile name and optional arguments: seed (seed for random number generation, know what is seed here), compiledir; directory for Theano complilation (Theano exclusive thing skippable in matlab) , recover; if True, will attempt to recover from a previously saved run, gpus; if gpus are to be used or not for the run, if it is greater than 0 then gpus will be used by theano

```
        # Theano setup
        os.environ.setdefault('THEANO_FLAGS', '')
        if compiledir is not None:
            os.environ['THEANO_FLAGS'] += ',base_compiledir=' + compiledir
        os.environ['THEANO_FLAGS'] += ',floatX=float32,allow_gc=False'
```

this part setups Theano flags i.e., tinkers with its configurations (Can be skipped in matlab)

the os.environ is part of os module and points to environment variables of the system, up above we are setting THEANO_FLAGS as empty string for now

then we check if compiledir is provided; if yes, set base compile dir as that compile dir

next Theano flags are added with floatx as float32: This sets the default dtype returned by tensor.matrix(), tensor.vector(), and similar functions of theano. It also sets the default Theano bit width for arguments passed as Python floating-point numbers and also speeds up Theano calculation,

allow_gc: This sets the default for the use of the Theano garbage collector for intermediate results. To use less memory, Theano frees the intermediate results as soon as they are no longer needed. Disabling Theano garbage collection allows Theano to reuse buffers for intermediate results between function calls. This speeds up Theano by no longer spending time reallocating space. This gives significant speed up on functions with many ops that are fast to execute, but this increases Theano's memory usage.

```
if gpus > 0:
            os.environ['THEANO_FLAGS'] += ',device=cuda'
```

Next if gpus is >0 then add gpu as the device to use for computing unfortunately there's some issues being encountered in using gpu in Theano. So, this code code amounts to no value as of now.

```
        # Only involve Theano for training
        from .trainer import Trainer
```

importing Trainer class from trainer.py

```
        # The task
        try:
            task = self.m.task
        except AttributeError:
            task = Struct(generate_trial=self.m.generate_trial)
```

try to keep self.m's task method in from of 'task' variable if error occurs then make a struct class object with generate trial as it's only function

for our S1_code case we will go down the except code route

```
        # Parameters
        params = {}
```

an empty parameter dictionary

```
        # Seed, if given
        if seed is not None:
            params['seed'] = seed
```

if seed given add it in params

```
        # Optional parameters
        for k in defaults:
            if hasattr(self.m, k):
                params[k] = getattr(self.m, k)
```

to know where defaults came from let's move over to top and find import statements

```
import importlib
import inspect
import os
import sys

import numpy as np

from .defaults import defaults, generate_trial
```

among various imports like importlib, inspect etc we see defaults and generate trial(this generate trial is just a dummy) from defaults.py file

let's take a look at default's file

```
"""
Default parameters for training.

"""
defaults = {
    'extra_info':           {},
    'Nin':                  0,
    'N':                    100,
    'Nout':                 1,
    'rectify_inputs':       True,
    'train_brec':           False,
    'brec':                 0,
    'train_bout':           False,
    'bout':                 0,
    'train_x0':             True,
    'x0':                   0.1,
    'mode':                 'batch',
    'tau':                  100,
    'tau_in':               100,
    'Cin':                  None,
    'Crec':                 None,
    'Cout':                 None,
    'ei':                   None,
    'ei_positive_func':     'rectify',
    'hidden_activation':    'rectify',
    'output_activation':    'linear',
    'n_gradient':           20,
    'n_validation':         1000,
    'gradient_batch_size':  None,
```

```
    'validation_batch_size': None,
    'lambda_Omega':          2,
    'lambda1_in':            0,
    'lambda1_rec':           0,
    'lambda1_out':           0,
    'lambda2_in':            0,
    'lambda2_rec':           0,
    'lambda2_out':           0,
    'lambda2_r':             0,
    'callback':              None,
    'performance':           None,
    'terminate':             (lambda performance_history: False),
    'min_error':             0,
    'learning_rate':         1e-2,
    'max_gradient_norm':     1,
    'bound':                 1e-20,
    'baseline_in':           0.2,
    'var_in':                0.01**2,
    'var_rec':               0.15**2,
    'seed':                  1234,
    'gradient_seed':         11,
    'validation_seed':       22,
    'structure':             {},
    'rho0':                  1.5,
    'max_iter':              int(1e7),
    'dt':                    None,
    'distribution_in':       None,
    'distribution_rec':      None,
    'distribution_out':      None,
    'gamma_k':               2,
    'checkfreq':             None,
    'patience':              None,
    'momentum':              False, # Not used currently
    'method':                'sgd'  # Not used currently
    }

def generate_trial(rng, dt, params):
    """
    Recommended structure for the `generate_trial` function.

    """
    pass
```

this was what default's file wholly comprised of the defaults imported from it was a dictionary and generate trial is just dummy function, a recommended structure for generate trial function

Wrapping back to model file:

```
        # Optional parameters
        for k in defaults:
            if hasattr(self.m, k):
                params[k] = getattr(self.m, k)
```

Now it will run a loop for every key ‘k’ in ‘defaults’ dictionary

For every iteration there will be a check if that key, as an attribute is present target model’s file(S1_code for our case; reminder self.m is a class object of Struct and possesses all attribute and function from modelfile) if it is then that key is added the same in params with key value extracted from self.m’s same matching attribute.

getattr() inbuilt function of python return’s attribute value from a class object. (refer here for more)

The params now:

```
Your params:  {'Nin': 2, 'N': 100, 'Nout': 2,
'Cout': array([[1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1.,
        1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1.,
        1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1.,
        1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1.,
        1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1.,
        0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.,
        0., 0., 0., 0.],
       [1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1.,
        1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1.,
        1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1.,
        1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1.,
        1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1.,
        0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.,
        0., 0., 0., 0.]]), 'ei': array([ 1,  1,  1,  1,  1,  1,  1,  1,  1,  1,
1,  1,  1,  1,  1,  1,  1,
        1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,
        1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,
        1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,
        1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1, -1, -1, -1, -1, -1,
       -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1]),
'n_validation': 1100, 'performance': <function performance_2afc at
0x0000026070B579D0>, 'terminate': <function terminate at 0x0000026070B5C9D0>}
```

```
        # Train
        trainer = Trainer(params)
        trainer.train(savefile, task, recover=recover)
```

Finally, we reach the last lines of the model.py file within these 2 lines, kickstarts the whole training regime of model

It makes ‘trainer’ named class object of Trainer class with params as argument

Moving over to Trainer class definition in trainer.py

```
class Trainer:
    """
    Train an RNN.

    """
    def __init__(self, params, floatX=theano.config.floatX):
        """
        Initialize.

        Parameters
        ----------

        params : dict
                 All parameters have default values, see `pycog.defaults`.

          Entries
          -------

          Nin : int, optional
                Number of input units.

          N : int, optional
              Number of recurrent units.

          Nout : int, optional
                 Number of output units.

          train_brec : bool, optional
                       Whether to train recurrent biases.

          brec : float, optional
                 Initial value of the recurrent bias.
```

```
        train_bout : bool, optional
                     Whether to train output biases.

        bout : float, optional
               Initial value of the output bias

        train_x0 : bool, optional
                   Whether to optimize the initial conditions.

        x0 : float, optional
             Initial value of the initial conditions.

        mode : str, optional
               `continuous` or `batch` training/running mode.

        tau : float or numpy.ndarray, optional
              Time constant(s) for recurrent units.

        Cin : numpy.ndarray or Connectivity, optional
              Input weight structure.

        Crec : numpy.ndarray or Connectivity, optional
               Recurrent weight structure.

        Cout : numpy.ndarray or Connectivity, optional
               Output weight structure.

        ei : numpy.ndarray, optional
             E/I signature.

        ei_positive_func : str, optional
                           Function to use to keep weights positive.

        hidden_activation : str, optional
                            Hidden activation function.

        output_activation: str, optional
                           Output activation function.

        n_gradient : int, optional
                     Minibatch size for gradient dataset.

        n_validation : int, optional
                       Minibatch size for validation dataset.
```

```
        gradient_batch_size, validation_batch_size : int, optional
                                                     Number of trials to
precompute
                                                     and store in each dataset.
Make
                                                     sure the batch sizes are
larger
                                                     than the minibatch sizes.

        lambda_Omega : float, optinonal
                       Multiplier for the vanishing gradient regularizer.

        lambda1_in, lambda1_rec, lambda1_out : float, optional
                                               Multipliers for L1 weight
regularization.

        lambda2_in, lambda2_rec, lambda2_out : float, optional
                                               Multipliers for L2 weight
regularization.

        lambda2_r : float, optional
                    Multiplier for L2 firing rate regularization.

        callback : function, optional
                   Evaluate validation dataset.

        performance : function, optional
                      Performance measure.

        terminate : function, optional
                    Custom termination criterion.

        min_error : float, optional
                    Target error. Terminate if error is less than or equal.

        learning_rate : float, optional
                        Learning rate for gradient descent.

        max_gradient_norm : float, optional
                            Clip gradient if its norm is greater than.

        bound : float, optional
                Lower bound for denominator in vanishing gradient regularizer.

        baseline_in : float, optional
```

```
                        Baseline input rate.

        var_in : float or numpy.ndarray, optional
                 Variance(s) for inputs.

        var_rec : float or numpy.ndarray, optional
                  If `float` or 1D `numpy.ndarray`, then recurrent units
receive
                  independent noise. If 2D `numpy.ndarray`, then noise is drawn
                  from a multivariable normal distribution.

        seed, gradient_seed, validation_seed : int, optional
                                               Seeds for the random number
generators.

        structure : dict, optional
                    Convey structure information, such as what each input
represents.

        rho0 : float, optional
               Spectral radius for the initial recurrent weight matrix.

        max_iter : int, optional
                   Maximum number of iterations for gradient descent.

        dt : float, optional
             Integration time step.

        distribution_in : str, optional
                          Distribution for the initial input weight matrix.

        distribution_rec : str, optional
                           Distribution for the initial recurrent weight
matrix.

        distribution_out : str, optional
                           Distribution for the initial output weight matrix.

        gamma_k : float, optional
                  k in Gamma(k, theta). Note mean = k*theta, var = k*theta^2.

        checkfreq : int, optional
                    Frequency with which to evaluate validation error.

        patience : int, optional
```

```
                Terminate training if the objective function doesn't change
                for longer than `patience`.

    Not used currently
    ------------------

    momentum, method

floatX : str, optional
        Floating-point type.

"""
```

NOTE: -Please spend some time reading above Comments as they are pretty valuable and defines most component variable to be used

The trainer class init function took params and floatx(optional Theano exclusive) arguments

```
self.p      = params.copy()
self.floatX = floatX
```

Now p attribute is created for the instance of class which has all values copied from params

Now the question might arise what is the difference between assigning using '=' and using '.copy()', the key difference is that when we '=' to assign, if we do some changes in right hand side item then the same changes will reflect lefth hand one too but when .copy() is used a copy of the item is assigned to left_side variable.

Same way floatx attribute is defined

```
#------------------------------------------------------------------------
# Fill in default parameters
#------------------------------------------------------------------------

# Default parameters
for k in defaults:
    self.p.setdefault(k, defaults[k])
```

a for loop is run for every key 'k' in defaults(same as in model.py) and the logic is that it kind of checks if the key is in params(obtained from model.py) or not if it is not, then this key will be inserted with value from defaults dict.

The setdefault() method returns the value of the item with the specified key. If the key does not exist, insert the key, with the specified value. (refer here)

Setdefault(<key>,<if_not_found_val_to_insert_with_that_key>)

Self.p now:

```
self.p is: {'Nin': 2, 'N': 100, 'Nout': 2,
'Cout': array([[1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1.,
        1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1.,
        1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1.,
        1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1.,
        1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1.,
        0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.,
        0., 0., 0., 0.],
       [1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1.,
        1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1.,
        1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1.,
        1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1.,
        1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1.,
        0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.,
        0., 0., 0., 0.]]), 'ei': array([ 1,  1,  1,  1,  1,  1,  1,  1,  1,  1,
1,  1,  1,  1,  1,  1,  1,
        1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,
        1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,
        1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,
        1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1, -1, -1, -1, -1, -1,
       -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1]),
'n_validation': 1100, 'performance': <function performance_2afc at
0x000001D4768A79D0>, 'terminate': <function terminate at 0x000001D4768AD9D0>,
'extra_info': {}, 'rectify_inputs': True, 'train_brec': False, 'brec': 0,
'train_bout': False, 'bout': 0, 'train_x0': True, 'x0': 0.1, 'mode': 'batch',
'tau': 100, 'tau_in': 100, 'Cin': None, 'Crec': None, 'ei_positive_func':
'rectify', 'hidden_activation': 'rectify', 'output_activation': 'linear',
'n_gradient': 20, 'gradient_batch_size': None, 'validation_batch_size': None,
'lambda_Omega': 2, 'lambda1_in': 0, 'lambda1_rec': 0, 'lambda1_out': 0,
'lambda2_in': 0, 'lambda2_rec': 0, 'lambda2_out': 0, 'lambda2_r': 0, 'callback':
None, 'min_error': 0, 'learning_rate': 0.01, 'max_gradient_norm': 1, 'bound': 1e-
20, 'baseline_in': 0.2, 'var_in': 0.0001, 'var_rec': 0.0225, 'seed': 1234,
'gradient_seed': 11, 'validation_seed': 22, 'structure': {}, 'rho0': 1.5,
'max_iter': 10000000, 'dt': None, 'distribution_in': None, 'distribution_rec':
None, 'distribution_out': None, 'gamma_k': 2, 'checkfreq': None, 'patience':
None, 'momentum': False, 'method': 'sgd'}
```

```
        # Time constants
        if not np.isscalar(self.p['tau']):
            self.p['tau'] = np.asarray(self.p['tau'], dtype=floatX)
```

Next up we check ‘tau’ is scalar or not in our p attribute (reason for this might be, not using default ‘tau’ value from ‘defaults’ as one is provided in modelfile)

If it’s not scalar then convert it into numpy array with datatype floatX()

Reason for this conversion is when provided is more than one value for ‘tau’, in form of list/array

Isscalar is a method(function) of numpy module and returns True when passed object is scalar

Asaarray converts any given input data that can be converted to array to numpy array without copying the data i.e. the data will not be copied and assigned specially for new array but will be used from parent object directly in other words any changes done on parent object will reflect over to new array. Refer here for more info about asarray (wondering about difference between asarray and array in numpy? Read here)

For S1_code ‘tau’ was not provided from modelfile and was extracted from defaults so the condition was never triggered for the case. default ‘tau’ is 100.

```
        # Time step
        if self.p['dt'] is None:
            if np.isscalar(self.p['tau']):
                self.p['dt'] = self.p['tau']/5
            else:
                self.p['dt'] = np.min(self.p['tau'])/5
```

timestep check:

‘dt’ represent time step i.e., time difference between every input point

Here it checks if ‘dt’ is present in params or not, if not then check whether ‘tau’ is scalar or not, if scalar then time step is one-fifth of time constant else ‘tau’ is not scalar and has proabaly multiple values stored in that case time step is one-fifth of minimum of time constant in ‘tau.

```
        # Distribution for initial weights (Win)
        if self.p['distribution_in'] is None:
            if self.p['ei'] is not None:
                self.p['distribution_in'] = 'uniform'
            else:
                self.p['distribution_in'] = 'uniform'
```

conforming probability distribution for input weights:

if input distribution is given in params then moves on, otherwise check if ‘ei’ (array of 1’s and -1’s created in S1_code file defining excitatory units and inhibitory units) is given or not, though regardless of the case they have assigned distribution as uniform.

```
        # Distribution for initial weights (Wrec)
        if self.p['distribution_rec'] is None:
            if self.p['ei'] is not None:
                self.p['distribution_rec'] = 'gamma'
            else:
                self.p['distribution_rec'] = 'normal'
```

conforming probability distribution for recurrent weights:

same logic as above just when ei is given distribution is gamma otherwise normal distribution.

```
        # Distribution for initial weights (Wout)
        if self.p['distribution_out'] is None:
            if self.p['ei'] is not None:
                self.p['distribution_out'] = 'uniform'
            else:
                self.p['distribution_out'] = 'uniform'
```

conforming probability distribution for recurrent weights:

Totally same logic as input weight distribution, assigned uniform in either case

For S1_code all of the three were None by default so these were triggered and as ‘ei’ was not None; Input distribution became uniform, Recurrent became gamma and output became normal distribution.

```
        # Default mask for recurrent weights
        if self.p['Crec'] is None:
            N = self.p['N']
            if self.p['ei'] is not None:
                # Default for E/I is fully (non-self) connected, mean-balanced
                exc, = np.where(self.p['ei'] > 0)
                inh, = np.where(self.p['ei'] < 0)

                C = np.zeros((N, N))
                for i in exc:
                    C[i,exc]  = 1
                    C[i,i]    = 0
                    C[i,inh]  = 1
                    C[i,inh] *= np.sum(C[i,exc])/np.sum(C[i,inh])
                for i in inh:
                    C[i,exc]  = 1
                    C[i,inh]  = 1
                    C[i,i]    = 0
                    C[i,inh] *= np.sum(C[i,exc])/np.sum(C[i,inh])
                C /= np.linalg.norm(C, axis=1)[:,np.newaxis]

                self.p['Crec'] = C
            else:
                # Default for no E/I is fully (non-self) connected
                C = np.ones((N, N))
                np.fill_diagonal(C, 0)

                self.p['Crec'] = C
```

If recurrent connectivity matrix is None i.e., not provided in params then:

Extract number of hidden layers from params and label it ‘N’

If ‘ei’ array is in params then:

Extract inhibitory(-1 places) and excitatory(+1 places) unit’s index from ei depending on where ever ei is >0 and <0 (recall where method of numpy, used in tasktools.py get_idx function) to form 2 arrays ‘exc’ for excitatory and ‘inh’ for inhibitory

Next make a matrix of size ‘N’ by ‘N’ label it as ‘C’

Now for every index ‘i’ in ‘exc’; at positions (i,exc) set as 1 i.e., position’s with row ‘i’ and all columns with index from ‘exc’ to 1, postion (i,i) as 0 (setting diagonal elements in excitatory part as 0) and similarly all positions (i,inh) 1 and Finally the positions (i,inh) are overridden with value of, sum of every value in C with row i and columns in exc divide by sum of every value in C with row i and columns in inh. [This iteration ran 80 times and modified first 80 rows]

Further for every index ‘i’ in ‘inh’; at positions (i,exc) set as 1 i.e., position’s with row ‘i’ and all columns with index from ‘exc’ to 1, similarly all positions (i,inh) 1, postion (i,i) as 0 (setting diagonal elements in whole matrix as 0) and Finally the positions (i,inh) are overridden with value of, sum of every value in C with row i and columns in exc divide by sum of every value in C with row i and columns in inh.

[This iteration ran 20 times and modified remaining 20 rows]

Lastly C is divided by 2d conversion of vector norm of it’s own matrix along column axis. In simpler words

first norm is taken for this C along column axis this is done by numpy.linalg.norm() method (linalg stands for linear algebra i.e., we are using numpy’s linear algebra functions from which we are specifically using norm method) the norm method takes the matrix as the primary argument and optionally axis at which to calculate norm by default it is None i.e. matrix norm is returned, but we are specifying it along for column axis (axis = 1) by specifying this we are extracting vector norm along the column axis,(refer here to know about this function more)

after this norm calculation we get a 1D matrix like this:

```
array([1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1.,
       1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1.,
       1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1.,
       1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1.,
       1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1.,
       1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1.])
```

Then by specifying) *‘[:,np.newaxis] ‘* as slicing index we are extrapolating a new axis into this 1D array making it 2D. by specifying newaxis at column position we are making it 2D with 2 columns and 100 rows as initially we had 100 lement array now we have a 100 element matrix with 100 rows and 2 columns with 1 column has original values and other is dummy i.e. is empty

See here about np.newaxis usage

The looks like this now

```
array([[1.],
       [1.],
       [1.],
       [1.],
       [1.],
       [1.],
       [1.],
       [1.],
       [1.],
       [1.],
       [1.],
       [1.],
       [1.],
       [1.],
       [1.],
       [1.],
```

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

```
       [1.],
       [1.],
       [1.],
       [1.],
       [1.],
       [1.],
       [1.],
       [1.],
       [1.],
       [1.],
       [1.],
       [1.],
       [1.],
       [1.],
       [1.],
       [1.],
       [1.],
       [1.],
       [1.],
       [1.],
       [1.],
       [1.],
       [1.],
       [1.],
       [1.],
       [1.],
       [1.],
       [1.],
       [1.],
       [1.],
       [1.],
       [1.],
       [1.],
       [1.],
       [1.],
       [1.],
       [1.],
       [1.],
       [1.],
       [1.],
       [1.]])
```

And finally C will be divided by this array

Then we just add this C in self.p (params copy variable) with key name as Crec standing for reccurent connectivity

Alas that's what changes in C were after iterations of for loop and normalization

```
C after first iteration:  [[0.   1.   1.   ... 3.95 3.95 3.95]
 [1.   0.   1.   ... 3.95 3.95 3.95]
 [1.   1.   0.   ... 3.95 3.95 3.95]
 ...
 [0.   0.   0.   ... 0.   0.   0.  ]
 [0.   0.   0.   ... 0.   0.   0.  ]
 [0.   0.   0.   ... 0.   0.   0.  ]]

C after second iteration:  [[0.         1.         1.         ... 3.95       3.95
3.95      ]
 [1.         0.         1.         ... 3.95       3.95       3.95      ]
 [1.         1.         0.         ... 3.95       3.95       3.95      ]
 ...
 [1.         1.         1.         ... 0.         4.21052632 4.21052632]
 [1.         1.         1.         ... 4.21052632 0.         4.21052632]
 [1.         1.         1.         ... 4.21052632 4.21052632 0.        ]]

C after normalization:  [[0.         0.05056894 0.05056894 ... 0.19974732
0.19974732 0.19974732]
 [0.05056894 0.         0.05056894 ... 0.19974732 0.19974732 0.19974732]
 [0.05056894 0.05056894 0.         ... 0.19974732 0.19974732 0.19974732]
 ...
 [0.04897948 0.04897948 0.04897948 ... 0.         0.20622941 0.20622941]
 [0.04897948 0.04897948 0.04897948 ... 0.20622941 0.         0.20622941]
 [0.04897948 0.04897948 0.04897948 ... 0.20622941 0.20622941 0.        ]]
```

(Values are being shown truncated here as, if these were to be shown fully it will take more than 50 pages to display any one of these, yes you read right 50 pages)

Back to our original motto, recall that we were in if statement condition which will run when ei is present in params. Now comes the case when ei is not present:

C will be matrix comprising of all values 1's of size N by N and its diagonal elements will be filled with 0 as shown, by numpy.fiill_diagonal function which do just as it says

And last but not least just like in if statement add this to self.p dict with key name Crec

```
        # Convert to connectivity matrices
        for k in ['Cin', 'Crec', 'Cout']:
            if self.p[k] is not None and not isinstance(self.p[k], Connectivity):
                self.p[k] = Connectivity(self.p[k])
```

Now last execution for __init__ of trainer class.

For every element ‘k’ in list ['Cin', 'Crec', 'Cout'] we check if this self.p[k] is not None (i.e. inside params do not have value None) and also is not an instance of connectivity class i.e. does not belong to class circle of connectivity. In those cases where these two condition hold true the self.p[k] will be overwritten with class object of connectivity with that same self.p[k] passed as argument

Note: if you take a look params from start you will see that Crec and Cin were None there but just along the previous few lines we changed Crec from None to a matrix with as calculated values hence these condition will hold False only for Cin which is still None.

So let’s see what this connectivity class is about and what happens to our supplies self.p[k]

```
class Connectivity:
    """
    Constrain the connectivity.

    """
```

Below part of is_connected will be skipped now and will be picked up back when called

```
@staticmethod
    def is_connected(C):
        """
        Return `True` if the square connection matrix `C` is connected, i.e.,
every
        unit is reachable from every other unit, otherwise `False`.

        Note
        ----

        This function only performs the check if the NetworkX package is
available:

          https://networkx.github.io/

        """
        if nx is None:
            return True

        G = nx.from_numpy_matrix(C, create_using=nx.DiGraph())
        return nx.is_strongly_connected(G)
```

```
    def __init__(self, C_or_N, Cfixed=None, p=1, rng=None, seed=4321):
        """
        Initialize.

        Parameters
        ----------

        C_or_N : 2D numpy.ndarray or int
                 If ``int``, create a random binary mask with density `p`.

        Cfixed : 2D numpy.ndarray
                 Fixed weights.

        p : float, optional
            Density of non-self connections.

        rng : numpy.random.RandomState, optional
              Random number generator.

        seed : int, optional
               Seed for random number generator if ``rng`` is not provided.

        """
```

The parameter self.p[k] we supplied before will take form of C_or_N here, other parameters will keep their default value intact to Know what these different parameter do read the comments

C_or_N can be either a numpy array(C) or simply an integer (N)

Our case is of C as we gave numpy array

In case it is N:

```
        if isinstance(C_or_N, int):
            N    = C_or_N
            ntot = N*(N-1)
            nnz  = int(p*ntot)

            if rng is None:
                rng = np.random.RandomState(seed)

            x = np.concatenate((np.ones(nnz, dtype=int), np.zeros(ntot-nnz,
dtype=int)))
            rng.shuffle(x)
```

```
            C = np.zeros((N, N), dtype=int)
            k = 0
            for i in range(N):
                for j in range(N):
                    if i == j: continue

                    C[i,j] = x[k]
                    k += 1
```

Checking if instance of int() function i.e. is an integer or not

If yes, N will be equal to C_or_N and ‘ntot’ will be N*(N-1) further ‘nnz’ will be integer value of density of non self connection times ‘ntot’

If rng is None i.e Not provided, then create numpy randomstate instance with given or default seed

Create x by concatenating a ones 1D matrix of size nnz datatype integer and zeroes matrix of size (ntot-nnz) datatype integer

Then random shuffle this matrix using rng

Make C matrix of zeroes of size N by N datatype integer

Assign k as 0

A loop runs for every i from range 0 to N-1; another loop for every iteration of first runs with every j from range 0 to N-1; set matrix C at position i,j as integer at index k of x and increment k by 1, if at any iteration i is equal to j then skip that iteration and move over to next iteration without running further

With this we will have a matrix C ready with random binary values with density p

Otherwise, if C_or_N is not affiliated with integer then Assign C simply as C_or_N.

```
        # Check that a square connection matrix is connected.
        if C.shape[0] == C.shape[1] and not Connectivity.is_connected(C):
            print("[ {}.Connectivity ] Warning: The connection matrix is not
connected.".format(THIS))
```

check if shape of C is of square matrix check if matrix is fully connected otherwise will give a warning

let’s explore .isconnectivity(C) to know how it checks this

```
    @staticmethod
    def is_connected(C):
        """
        Return `True` if the square connection matrix `C` is connected, i.e.,
every
        unit is reachable from every other unit, otherwise `False`.

        Note
        ----

        This function only performs the check if the NetworkX package is
available:

          https://networkx.github.io/

        """
        if nx is None:
            return True

        G = nx.from_numpy_matrix(C, create_using=nx.DiGraph())
        return nx.is_strongly_connected(G)
```

@staticmethod defines the is_connected function as static method i.e. a function that doesn't use self argument making it a function independent of state of object itself

for more detail info about @staticmethod refer here (presumably it has no significant use being here, probably just so they don't need to specify self here)

So starting out, if nx is None (will happen when NetworkX package is not available) in those cases return True as it can't check the connectivity so no point in returning False an giving false warning to user about connectivity

Note: for every model that was trained by us in practical, we didn't installed Networkx so we usually skipped through this section hence it can be concluded that it is not of immediate importance and can be ignored as long as we are using recommended way for creating 'C'

networkx import statement:

```
try:
    import networkx as nx
except ImportError:
    nx = None
```

network.from_numpy_matrix() function returns a graph from numpy matrix.

The numpy matrix is interpreted as an adjacency matrix for the graph.

Create_using argument specifies graph type to create. If graph instance, then cleared before populated.

Refer here for more info

Next the return statement tests directed graph for strong connectivity.

A directed graph is strongly connected if and only if every vertex in the graph is reachable from every other vertex.

If it is then returns True otherwise False

Depending on True or False user will get a warning or not

```
self.define(C, Cfixed)
```

final call by connectivity.py init function

calling define function of the same class by using self and passing C and Cfixed as argument

Moving over to define function

```
    def define(self, C, Cfixed):
        """
        C : 2D numpy.ndarray
            Plastic weights

        Cfixed : 2D numpy.ndarray
                 Fixed weights
```

Function that defines connecti

```
        """
        # Connectivity properties
        self.C     = C
        self.shape = C.shape
        self.size  = C.size
```

Extracting connectivity properties

Making instance variable of C, its shape and size

```
        # Plastic weights
        self.plastic      = C[np.where(C != 0)]
        self.nplastic     = len(self.plastic)
        self.idx_plastic, = np.where(C.ravel() != 0)

      self.mask_fixed = w.reshape(C.shape)
```

Defining plastic weights i.e., weights that may change throughout training

Making instance variable plastic with values that are not zero present in C

Extracting the length of this plastic instance variable

Extracting Indexes from flattened C where ever it's element are not zero (You might be wondering what is the difference flattening made here; the C was 100 by 100 matrix in start by flattening it became 1000 element 1D matrix)

numpy.ravel(): Flattens array/matrix passed to it

```
        # Fixed weights
        if Cfixed is not None:
            self.fixed      = Cfixed[np.where(Cfixed != 0)]
            self.nfixed     = len(self.fixed)
            self.idx_fixed, = np.where(Cfixed.ravel() != 0)
        else:
            self.fixed     = np.zeros(0)
            self.nfixed    = 0
            self.idx_fixed = np.zeros(0, dtype=int)
```

Time to define Fixed weights that won't budge a bit throughout training.

If Cfixed is given which will be an array contatining zeroes and fixed weights value at their excat position

So, if we have this array then:

Create instance variable 'fixed' which will hold all fixed values from Cfixed

Then extracting Number of the fixed weights from this fixed

And finally assigning index of all these fixed from flattened matrix of Cfixed

If Cfixed is none then:

Create 'fixed' instance variable with zero matrix of 0 size similarly number of fixed will be zero and index matrix will be also zero array matrix.

For our case Cfixed was None, So we didn't have any connectivity fixed weights

```
        # Check that indices do not overlap
        assert len(np.intersect1d(self.idx_plastic, self.idx_fixed)) == 0
```

Does the same what comment says

Recall assert checks if the condition provided holds if not then rams the execution in assertion error

```
        self.p_plastic = self.nplastic/self.size
        self.p         = (self.nplastic + self.nfixed)/self.size
```

creating instance variable ‘p_plastic’ that will hold probability/density of plastic values as clear from code it is division operation between number of plastic values ‘nplastic’ and size of C (connectivity matrix)

next total density of all values plastic and fixed alike. First intuition may direct you into assuming it will be one but take a look back to condition for plastic and fixed values, we have actually skipped all zero values from connectivity matrix so this p will be slightly less than 1.

```
Your self.p is : 0.99
```

(Caution: Do not confuse or lose track by mixing this self.p with one denoting parameters in trainer class we are currently in connectivity class where we have no contact with that self.p)

Moving on

```
        # Mask for plastic weights
        w = np.zeros(C.size)
        w[self.idx_plastic] = 1
        self.mask_plastic = w.reshape(C.shape)
```

creating mask for plastic weights (to know what mask is go through paper’s ‘Specifying the pattern of connectivity’ under ‘Materials and Methods’)

creating w named variable that will be a zero matrix of size length same as C(there a confusion may arise w and C **don’t** have same **shape** by have; C is a 2D matrix of shape 100 by 100 size is 10000 and w is a 1D matrix with size and shape both 10000)

next at every index from plastic index variable we created before set value as 1 (now you might see the point in flattening the C before extracting indexes; regardless the logic is that take out index from plastic matrix and set those index as 1 in w)

then we reshape w in shape of C i.e., 100 by 100

For our case mask_plastic is one everywhere except places with zero

```
Mask_plastic
array([[0., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1.,
        1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1.,
        1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1.,
        1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1.,
        1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1.,
        1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1.,
        1., 1., 1., 1.],
       [1., 0., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1.,
        1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1.,
        1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1.,
        1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1.,
        1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1.,
        1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1.,
        1., 1., 1., 1.],...
      ...
      ...
      ...
       ...[1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1.,
        1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1.,
        1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1.,
        1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1.,
        1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1.,
        1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1.,
        1., 1., 1., 0.]])
```

```
        # Mask for fixed weights
        w = np.zeros(C.size)
        w[self.idx_fixed] = self.fixed
        self.mask_fixed = w.reshape(C.shape)
```

Same logic as mask for plastic values

For our case it is all zero

With this we have covered everything that happens in connectivity.py

Summary:

So we have a loop in trainer that will run 3 times for (Cin, Cout, Crec) and overrite their values in params with connectivity class objects of these as argument to the class.

Then Connectivity class's init function will trigger resulting in connectivity checks and processing further C to baton pass to 'define' function of same class where the mask for plastic and fixed values will be defined.

So, we are done with all executions in connectivity.py file coming back to trainer we see the end of it's init excetution as well

So, Move over back to model.py

And start execution of next line

```
        trainer.train(savefile, task, recover=recover)
```

if you have followed correctly then now we are with trainer object created pages ago

Now we call train method of trainer class,

Time to go back to Trainer class,

```
    def train(self, savefile, task, recover=True):
        """
        Train the RNN.

        Parameters
        ----------

        savefile : str

        task : function

        recover : bool, optional
                  If `True`, will attempt to recover from a previously saved run.

        """
```

It takes name of savefile, task(generate trial function) and if to recover from previous run a 'recover' argument

```
        N     = self.p['N']
        Nin   = self.p['Nin']
        Nout  = self.p['Nout']
        alpha = self.p['dt']/self.p['tau']
```

assignment of instance variables we declared in previously in trainer __init__ to new variable names for ease of use

```
        # Initialize settings
        settings = OrderedDict()
```

A normal dictionary has a certain property that it's key- value pair has no order that's where an ordered dictionary come into picture it is a special subclass of dictionary which remembers the order of key-value pair insertion. To know more about it refer here

We create a settings variable which is an empty ordered dictionary.

This setting variable will also be used to display the parameters and their values that we see when we train a model.

```
        # Check if file already exists
        if not recover:
            if os.path.isfile(savefile):
                os.remove(savefile)
```

Now the if condition here works reverse of what recover is (if recover=True then if condition = False and vice versa if condition will be True for the other case)

So if, if condition is True then that means we do not have to recover from previous run hence we will clean up any previous file run if present.

os module allows one to run commands related to os tasks like meddling with files, running cmd commands etc. here we used .path.isfile() method that checks if a file is present at that location or not, and .remove() method that removes a file from the path specified

(Now you might wonder when we run the file it gives error when a savefile is already present, so why does it not clean that up? The answer is 'recover' by default has value True hence it never tries to clean up old data before a new run)

```
        #--------------------------------------------------------------------
        # Are we using GPUs?
        #--------------------------------------------------------------------

        if theanotools.get_processor_type() == 'gpu':
            settings['GPU'] = 'enabled'
        else:
            settings['GPU'] = 'no'
```

Just does what it says; if Theano is using GPU then ‘enabled’ string value added with key ’GPU’ in settings dict otherwise ‘no’ value with that key

```
        #--------------------------------------------------------------------
        # Random number generator
        #--------------------------------------------------------------------

        settings['init seed'] = self.p['seed']
        rng = np.random.RandomState(self.p['seed'])
```

adding key ‘init seed’ in setting dict with seed value from our resourceful self.p’s seed attribute

initializing our random number generator with initial seed from params and store variable as ‘rng’

```
        #--------------------------------------------------------------------
        # Weight initialization
        #--------------------------------------------------------------------

        settings['distribution (Win)']  = self.p['distribution_in']
        settings['distribution (Wrec)'] = self.p['distribution_rec']
        settings['distribution (Wout)'] = self.p['distribution_out']

        if Nin > 0:
            Win_0 = self.init_weights(rng, self.p['Cin'], N, Nin,
                                      self.p['distribution_in'])
        Wrec_0 = self.init_weights(rng, self.p['Crec'],
                                   N, N, self.p['distribution_rec'])
        Wout_0 = self.init_weights(rng, self.p['Cout'],
                                   Nout, N, self.p['distribution_out'])
```

Then adding more keys to setting dictionary about distribution of input, output and recurrent weights

Moving on if number are inputs are not zero the initialize weights using ‘init_weights’ function of the same class Trainer

We supply rng, self.p[Cin], N, Nin, self.p['distribution_in’] as argument to that function

Scrolling over to init_weights function definition and see what happens next

```
    def init_weights(self, rng, C, m, n, distribution):
        """
        Initialize weights from a distribution.

        Parameters
        ----------

        rng : numpy.random.RandomState
              Random number generator.

        C : Connectivity
            Specify which weights are plastic and nonzero.

        m, n : int
               Number of rows and columns, respectively.

        distribution : str
                       Name of the distribution.

        """
```

Read above comments ^

```
        # Account for plastic and fixed weights.
        if C is not None:
            mask = C.plastic
            size = C.nplastic
        else:
            mask = 1
            size = m*n
```

Now Cin is None so we will go through with mask being 1 and m*n size

For case to come when C will be Crec and Cout, it will not be None so mask and size of them will be extracted from them(recall that this C is an object of connectivity class where we individually once defined plastic mask and their sizes for Crec and Cout)

```
        # Distributions
        if distribution == 'uniform':
            w = 0.1*rng.uniform(-mask, mask, size=size)
        elif distribution == 'normal':
            w = rng.normal(np.zeros(size), mask, size=size)
        elif distribution == 'gamma':
            k     = self.p['gamma_k']
            theta = 0.1*mask/k
            w     = rng.gamma(k, theta, size=size)
        elif distribution == 'lognormal':
            mean  = 0.5*mask
            var   = 0.1
            mu    = np.log(mean/np.sqrt(1 + var/mean**2))
            sigma = np.sqrt(np.log(1 + var/mean**2))
            w     = rng.lognormal(mu, sigma, size=size)
        else:
            raise NotImplementedError("[ {}.Trainer.init_weights ] distribution:
{}"
                                      .format(THIS, distribution))
```

A conclusion: 4 types of weight distribution are possible; uniform, normal, gamma, lognormal

Initialize weights according to the distribution:

If normal distribution draws a random uniform distribution of size from connectivity matrix and low as -mask and high as +mask lastly multiplies it by 0.1. know about rng.uniform here

If Distribution is normal then using a rng.normal draw random samples from a normal (Gaussian) distribution. With 1st argument denoting centres of the distribution, 2nd argument standard deviation (spread or “width”) of the distribution and last argument the size of the distribution. know more here

If distribution is gamma, take out gamma_k value from params(params got this from defaults for case of S1_code) as , theata variable will be 0.1 times mask dividend by k (recall mask is a numpy array and supports vector operations) finally supply these two along with size to rng.gamma method that will draw gamma distribution from them. The k is called shape of the gamma distribution and theta is called scale; both must be non-negative. default scale for rng.gamma method is 1. know more here

If the distribution is lognormal then get mean as 0.5 times mask, variance as 1, mu (mean value of underlying normal distribution) as log of (mean divide by square root of (1 +variance divide by mean square)) and sigma is square root of(log of(1 +variance divide by mean square)). Finally supply these to lognormal method and we will get our weight distributions

If none of the above is the required distribution then raise an error and end the execution then and there

w by far for inp, out and rec distribution:

```
Your input w is :  [-0.06169611  0.02442175 -0.01245445  0.05707172  0.05599516 -
0.04548148
 -0.04470715  0.06037444  0.09162787  0.07518653 -0.02843655  0.00019903
  0.03669259  0.04254041 -0.02594985  0.01223924  0.00061663 -0.09724631
  0.05456532  0.07652824 -0.0270228   0.02307924 -0.08492375 -0.0262352
  0.08662802  0.03027563 -0.02055948  0.05774603 -0.03663278  0.01361973
  0.07382548 -0.01276532  0.06042953 -0.07124664  0.04085219  0.04091626
 -0.05624158  0.08497353 -0.01157185  0.08186319 -0.08803816 -0.06314258
 -0.09052894  0.03497619  0.01892496  0.00666203 -0.09133519  0.01228662
 -0.03406631  0.00059337 -0.07762114  0.02143874  0.01318893 -0.09864719
  0.02348834  0.08242458  0.05810483  0.09841629  0.09176035  0.05839283
 -0.04294981  0.02498334 -0.00438124 -0.06086496 -0.02353651 -0.08922526
 -0.00967032  0.09640095 -0.07521146 -0.07612382  0.04770461  0.01746073
 -0.00567349 -0.07857464 -0.05415629  0.07999304 -0.01664929  0.00717033
 -0.0987583  -0.03987166 -0.01262137  0.0224298   0.08363962  0.02514733
  0.04119951 -0.07003326  0.04921268  0.0662014   0.02674515 -0.01233802
 -0.06948545  0.01368192  0.00564486  0.09028575 -0.00392816  0.00051191
  0.00737564  0.06384041 -0.08857687  0.03388435  0.05342333  0.04162307
  0.05937344  0.01155217  0.09316731 -0.07056862 -0.0940706   0.0187787
 -0.07718686  0.09016197 -0.03485852 -0.06127626 -0.00843767  0.08408051
  0.07581383 -0.04947685 -0.03039824 -0.06348225  0.08035921  0.04130563
  0.04533169  0.08001757  0.05583276  0.01983096 -0.04177495 -0.06972095
 -0.03296507  0.03151036 -0.08533149 -0.08899872 -0.03536104  0.01809636
  0.07077971 -0.04258751 -0.06538655 -0.07319576  0.09893077 -0.06410043
 -0.03649064  0.01365828 -0.09813029  0.08012972  0.09544829  0.01137894
 -0.08304523 -0.03339951  0.04568574 -0.07151293  0.01049379 -0.04539135
  0.09489903  0.03355738 -0.04886934 -0.0783377   0.05523614  0.0564956
  0.05232078  0.08288062  0.03172456  0.01367352 -0.05964886  0.03965928
  0.09043908  0.07799266  0.09871347  0.0637407   0.00902443 -0.00974919
  0.07811144  0.09465296  0.01868227 -0.0267851  -0.03538106  0.07428465
 -0.05687319  0.04698904 -0.02687618  0.06032052  0.05654712  0.04027108
  0.02455532 -0.00126347  0.06810754  0.0424194  -0.0112182  -0.09379303
 -0.02735205  0.04614436 -0.00488669 -0.03111661  0.02817609 -0.07475894
 -0.06570695  0.0474173  -0.07459412 -0.02607003  0.0208668  -0.07937911
  0.06047484  0.08911065]
Your rec w is :  [0.00661778 0.00485262 0.00398149 ... 0.03439099 0.03946075
0.00652401]
Your output w is :  [-0.03608214  0.03343297 -0.04045303 -0.00865651  0.08375766
0.00585763
  0.03374115 -0.0641578   0.0211406   0.00138595  0.03517649 -0.03900956
  0.05561343  0.02810439 -0.01406633  0.02415343  0.05406051 -0.02688903
  0.03091859 -0.07056442 -0.0742842   0.02606742  0.07790615  0.02538231
 -0.03250308 -0.08562085 -0.0977301  -0.04848051 -0.06427759 -0.00596862
```

```
 -0.03007779  0.03276129 -0.09603656 -0.01822404 -0.01211061  0.03370667
  0.06686558  0.00045424  0.01513263 -0.09735836 -0.07973206  0.00724779
 -0.0103606  -0.04793091  0.00604363 -0.02379264 -0.0716166   0.02640597
 -0.01947684  0.08903674 -0.05449918 -0.02460082  0.01686713  0.05694358
  0.0557398   0.00797183 -0.04906545  0.01173353 -0.09413288  0.0482416
  0.02711536 -0.04333581  0.02215264 -0.03132125  0.0857948  -0.09766495
 -0.08218586 -0.07940644 -0.01432988 -0.05722216 -0.00193008 -0.06628707
  0.0818024  -0.07544459  0.03141144  0.00749048 -0.02801166 -0.04068119
  0.07362721  0.08320969 -0.03484885  0.04814097 -0.07121574  0.01720465
  0.05618819 -0.01346189  0.00231432  0.08794512 -0.06330081  0.07230752
 -0.09574525 -0.00787106 -0.01987687 -0.06570112 -0.08920009  0.09492759
 -0.03289591  0.08658624 -0.05320484  0.06266279 -0.08572746  0.02111571
  0.03608305 -0.05599679  0.05016205  0.07729683  0.02823669 -0.07464613
 -0.09589763  0.04874223 -0.07218105 -0.0988131  -0.00612792  0.01467393
 -0.09385128  0.00474076 -0.07647422 -0.02622559 -0.03945378  0.0584092
  0.09570099  0.06901172 -0.00873453  0.08409824 -0.07903768 -0.02054283
  0.04659367 -0.07400037  0.09376829  0.02586716 -0.09894017 -0.02656973
 -0.09730869 -0.09128317  0.06646297 -0.02427635  0.06629728  0.01186819
 -0.00449774 -0.09634427 -0.03192018 -0.03881574  0.02958845 -0.00380478
  0.01755075 -0.03909087 -0.06699134  0.02170532  0.074958    0.05481193
 -0.01822937 -0.09566006  0.0821432  -0.02407616  0.08185078 -0.00804539
  0.05474689 -0.07499367  0.0319978   0.05745263]
```

(rec w may look small at first glance but is largest one and is shown truncated here)

```
        if C is not None:
            W = np.zeros(m*n)
            W[C.idx_plastic] = w
        else:
            W = w
```

If connectivity matrix is not None(Crec and Cout) then create a ‘W’ variable as zero matrix of size m times n and assign all plastic indexes with distribution values

Otherwise W is directly equal to drawn distribution (Cin)

W:

```
Your input W is :  [-0.06169611  0.02442175 -0.01245445  0.05707172  0.05599516 -
0.04548148
 -0.04470715  0.06037444  0.09162787  0.07518653 -0.02843655  0.00019903
  0.03669259  0.04254041 -0.02594985  0.01223924  0.00061663 -0.09724631
  0.05456532  0.07652824 -0.0270228   0.02307924 -0.08492375 -0.0262352
  0.08662802  0.03027563 -0.02055948  0.05774603 -0.03663278  0.01361973
  0.07382548 -0.01276532  0.06042953 -0.07124664  0.04085219  0.04091626
 -0.05624158  0.08497353 -0.01157185  0.08186319 -0.08803816 -0.06314258
```

```
 -0.09052894  0.03497619  0.01892496  0.00666203 -0.09133519  0.01228662
 -0.03406631  0.00059337 -0.07762114  0.02143874  0.01318893 -0.09864719
  0.02348834  0.08242458  0.05810483  0.09841629  0.09176035  0.05839283
 -0.04294981  0.02498334 -0.00438124 -0.06086496 -0.02353651 -0.08922526
 -0.00967032  0.09640095 -0.07521146 -0.07612382  0.04770461  0.01746073
 -0.00567349 -0.07857464 -0.05415629  0.07999304 -0.01664929  0.00717033
 -0.0987583  -0.03987166 -0.01262137  0.0224298   0.08363962  0.02514733
  0.04119951 -0.07003326  0.04921268  0.0662014   0.02674515 -0.01233802
 -0.06948545  0.01368192  0.00564486  0.09028575 -0.00392816  0.00051191
  0.00737564  0.06384041 -0.08857687  0.03388435  0.05342333  0.04162307
  0.05937344  0.01155217  0.09316731 -0.07056862 -0.0940706   0.0187787
 -0.07718686  0.09016197 -0.03485852 -0.06127626 -0.00843767  0.08408051
  0.07581383 -0.04947685 -0.03039824 -0.06348225  0.08035921  0.04130563
  0.04533169  0.08001757  0.05583276  0.01983096 -0.04177495 -0.06972095
 -0.03296507  0.03151036 -0.08533149 -0.08899872 -0.03536104  0.01809636
  0.07077971 -0.04258751 -0.06538655 -0.07319576  0.09893077 -0.06410043
 -0.03649064  0.01365828 -0.09813029  0.08012972  0.09544829  0.01137894
 -0.08304523 -0.03339951  0.04568574 -0.07151293  0.01049379 -0.04539135
  0.09489903  0.03355738 -0.04886934 -0.0783377   0.05523614  0.0564956
  0.05232078  0.08288062  0.03172456  0.01367352 -0.05964886  0.03965928
  0.09043908  0.07799266  0.09871347  0.0637407   0.00902443 -0.00974919
  0.07811144  0.09465296  0.01868227 -0.0267851  -0.03538106  0.07428465
 -0.05687319  0.04698904 -0.02687618  0.06032052  0.05654712  0.04027108
  0.02455532 -0.00126347  0.06810754  0.0424194  -0.0112182  -0.09379303
 -0.02735205  0.04614436 -0.00488669 -0.03111661  0.02817609 -0.07475894
 -0.06570695  0.0474173  -0.07459412 -0.02607003  0.0208668  -0.07937911
  0.06047484  0.08911065]
Your rec W is :  [0.         0.00661778 0.00485262 ... 0.03946075 0.00652401 0.
]
Your out W is :  [-0.03608214  0.03343297 -0.04045303 -0.00865651  0.08375766
0.00585763
  0.03374115 -0.0641578   0.0211406   0.00138595  0.03517649 -0.03900956
  0.05561343  0.02810439 -0.01406633  0.02415343  0.05406051 -0.02688903
  0.03091859 -0.07056442 -0.0742842   0.02606742  0.07790615  0.02538231
 -0.03250308 -0.08562085 -0.0977301  -0.04848051 -0.06427759 -0.00596862
 -0.03007779  0.03276129 -0.09603656 -0.01822404 -0.01211061  0.03370667
  0.06686558  0.00045424  0.01513263 -0.09735836 -0.07973206  0.00724779
 -0.0103606  -0.04793091  0.00604363 -0.02379264 -0.0716166   0.02640597
 -0.01947684  0.08903674 -0.05449918 -0.02460082  0.01686713  0.05694358
  0.0557398   0.00797183 -0.04906545  0.01173353 -0.09413288  0.0482416
  0.02711536 -0.04333581  0.02215264 -0.03132125  0.0857948  -0.09766495
 -0.08218586 -0.07940644 -0.01432988 -0.05722216 -0.00193008 -0.06628707
  0.0818024  -0.07544459  0.03141144  0.00749048 -0.02801166 -0.04068119
  0.07362721  0.08320969  0.          0.          0.          0.
  0.          0.          0.          0.          0.          0.
```

```
 0.          0.          0.          0.          0.          0.
 0.          0.          0.          0.         -0.03484885  0.04814097
-0.07121574  0.01720465  0.05618819 -0.01346189  0.00231432  0.08794512
-0.06330081  0.07230752 -0.09574525 -0.00787106 -0.01987687 -0.06570112
-0.08920009  0.09492759 -0.03289591  0.08658624 -0.05320484  0.06266279
-0.08572746  0.02111571  0.03608305 -0.05599679  0.05016205  0.07729683
 0.02823669 -0.07464613 -0.09589763  0.04874223 -0.07218105 -0.0988131
-0.00612792  0.01467393 -0.09385128  0.00474076 -0.07647422 -0.02622559
-0.03945378  0.0584092   0.09570099  0.06901172 -0.00873453  0.08409824
-0.07903768 -0.02054283  0.04659367 -0.07400037  0.09376829  0.02586716
-0.09894017 -0.02656973 -0.09730869 -0.09128317  0.06646297 -0.02427635
 0.06629728  0.01186819 -0.00449774 -0.09634427 -0.03192018 -0.03881574
 0.02958845 -0.00380478  0.01755075 -0.03909087 -0.06699134  0.02170532
 0.074958    0.05481193 -0.01822937 -0.09566006  0.0821432  -0.02407616
 0.08185078 -0.00804539  0.05474689 -0.07499367  0.0319978   0.05745263
 0.          0.          0.          0.          0.          0.
 0.          0.          0.          0.          0.          0.
 0.          0.          0.          0.          0.          0.
 0.          0.        ]
```

```
        return W.reshape((m, n))
```

finally return the ‘W’ by reshaping it to desired m by n size

The same explanation will suffice for Crec and Cout in next 2 lines of code

Thus, ending weight initialization

```
        #-----------------------------------------------------------------------
        # Enforce Dale's law on the initial weights
        #-----------------------------------------------------------------------

        settings['Nin/N/Nout'] = '{}/{}/{}'.format(Nin, N, Nout)

        if self.p['ei'] is not None:
            Nexc = len(np.where(self.p['ei'] > 0)[0])
            Ninh = len(np.where(self.p['ei'] < 0)[0])
            settings['Dale\'s law'] = 'E/I = {}/{}'.format(Nexc, Ninh)

            if Nin > 0:
                Win_0 = abs(Win_0) # If Dale, assume inputs are excitatory
            Wrec_0 = abs(Wrec_0)
            Wout_0 = abs(Wout_0)
        else:
            settings['Dale\'s law'] = 'no'
```

Adding Nin, N and Nout values to settings

If ei is present enforce Date's law

Take out number of excitatory and inhibitory unit count by taking length of an array where values are 1 for excitatory in ei and values are -1 for inhibitory in ei

Then add this to setting as the E/I is Nexc/Ninh (which apparently for our case is 80/20)

If Nin> 0 then convert input weight matrix(recall we got it from init_weight) as absolute of itself i.e., negate the negatives. Same will be done for Wrec_0 and Wout_0

Other case when ei is none just add dale's law keys in settings dict as value 'no'

```
        #--------------------------------------------------------------------------
        # Fix spectral radius
        #--------------------------------------------------------------------------
        # Compute spectral radius
        C = self.p['Crec']
        if C is not None:
            Wrec_0_full = C.mask_plastic*Wrec_0 + C.mask_fixed
        else:
            Wrec_0_full = Wrec_0
        if self.p['ei'] is not None:
            Wrec_0_full = Wrec_0_full*self.p['ei']
        rho = RNN.spectral_radius(Wrec_0_full)
```

assign Crec as C

check if C is not None if not none make a new variable 'Wrec_0_full' that will be Weight of recurrent matrix times mask of recurrent matrix plus fixed weight matrix (refer to paper to know about this equation)

if C_rec is None then assign Wrec_0_full directly as Wrec_0

next up check if ei is none or not if it is then covert Wrec_0_full as it's multiplication with ei array i.e., assigning excitatory and inhibitatory units

next supply this Wrec_0_full to RNN class's spectral radius function which will give rho as return.

Heading over to rnn.py file's spectral radius definition:

```
class RNN:
    """
    Recurrent neural network.

    """
    defaults = {
        'threshold': 1e-4,
        'sigma0':    0
        }
    ou_defaults = {
        'N':                 100,
        'Nin':               0,
        'Nout':              0,
        'hidden_activation': 'linear',
        'output_activation': 'linear',
        'baseline_in':       0,
        'rectify_inputs':    False,
        'var_in':            0.01**2,
        'var_rec':           0.15**2,
        'dt':                0.5,
        'tau':               100,
        'mode':              'batch'
        }
    dtype = np.float32
```

Some default parameters defined above

```
   @staticmethod
    def spectral_radius(A):
        """
        Compute the spectral radius of a matrix.

        """
        return np.max(abs(np.linalg.eigvals(A)))
```

This function is also a static method (i.e., don't require self can be called from class without class object)

It returns the spectral radius for input matrix A it does so by calculating eigen values for A then making them absolute and finally returning the max of them.

```
Your rho is :  0.10452500852748517
```

```
        # Scale Wrec to have fixed spectral radius
        if self.p['ei'] is not None:
            R = self.p['rho0']/rho
        else:
            R = 1.1/rho
        Wrec_0 *= R
        if C is not None:
            C.mask_fixed *= R
```

Above is mostly self-explanatory and logic had been already explained before

Wrec_0*=R will same as Wrec_0 =Wrec_0*R

```
        # Check spectral radius
        if C is not None:
            Wrec_0_full = C.mask_plastic*Wrec_0 + C.mask_fixed
        else:
            Wrec_0_full = Wrec_0
        if self.p['ei'] is not None:
            Wrec_0_full = Wrec_0_full*self.p['ei']
        rho = RNN.spectral_radius(Wrec_0_full)
        settings['initial spectral radius'] = '{:.2f}'.format(rho)
```

if guide till now is followed correctly you will automatically know now what component is doing what by taking a look at above code

From now guide will focus and explaining anything new and significant coming while skipping minor detail and leaving them for reader to understand from code (like if-else statement, assignments etc.)

First 4 lines are way similar to ‘compute spectral radius’ code just one new line that adds new key for spectral radius in settings (Something feels weird? Why same code 2 times!)

{:,2f} formats the integer till 2 decimal places at the placeholder space

```
        #-----------------------------------------------------------------------
        # Others
        #-----------------------------------------------------------------------

        brec_0 = self.p['brec']*np.ones(N)
        bout_0 = self.p['bout']*np.ones(Nout)
        x0_0   = self.p['x0']*np.ones(N)
```

Just assignments making ones matrix of specified sizes then multiplying them by specified number to get matrix with all values as that specific number. Keeps these assignments at back of mind to understand further code.(brec and bout are biase matrix for reccurent and output while x0_0 is Initial value of the initial conditions.)

```
        #----------------------------------------------------------------------
        # RNN parameters
        #----------------------------------------------------------------------

        if Nin > 0:
            Win = theanotools.shared(Win_0, name='Win')
        else:
            Win = None
        Wrec = theanotools.shared(Wrec_0, name='Wrec')
        Wout = theanotools.shared(Wout_0, name='Wout')
        brec = theanotools.shared(brec_0, name='brec')
        bout = theanotools.shared(bout_0, name='bout')
        x0   = theanotools.shared(x0_0,   name='x0')
```

these now calls the theanotools shared function

Going over to theanotools shared function definition

```
def shared(x, dtype=theano.config.floatX, **kwargs):
    if x.dtype == dtype:
        return theano.shared(x, **kwargs)
    return theano.shared(np.asarray(x, dtype=dtype), **kwargs)
```

arguments are x; a matrix, datatype, and keyworded variable-length argument list **kwargs

in our case x will be Wrec, Wout, brec etc., dtype will be default and **kwargs will be 'name=*<name>*' argument

then it checks if the datatype of x is same as datatype it needs i.e. theano floatx if yes then make a shared variable in Theano with name provided in **kwargs and x as the value or data of that variable and return it

in other case where the dtype is not same it makes a numpy array out of x with dtype specified as the one needed or specified by argument 'dtype' and same process of making and returning Theano shared variable will commence just like in if statement.

The whole process will repeat for Wrec, Wout, brec, bout and x0, and these 6 will now become Theano shared variables.

A shared variable are one which are shared between different functions and also between multiple calls to the same function. To cite an example, while training a neural network you create weights vector for assigning a weight to each feature under consideration. This vector is modified on every iteration during the network training. Thus, it has to be globally accessible across the multiple calls to the same function. So we create a shared variable for this purpose

Refer here for more about shared function of Theano

Shared variable helps in simplifying the operations over a pre-defined variable. for example, suppose we want to add two matrix, let's say a and b, where the value of b matrix will remain constant throughout the lifetime of the program, we can have the b matrix as the shared variable and do the required operation.

Code without using shared variable:

```
a = theano.tensor.matrix('a')
b = theano.tensor.matrix('b')
c = a + b
f = theano.function(inputs = [a, b], outputs = [c])
output = f([[1, 2, 3, 4]], [[5, 5, 6, 7]])
```

Code using shared variable :

```
a = theano.tensor.matrix('a')
b = theano.tensor.shared( numpy.array([[5, 6, 7, 8]]))
c = a + b
f = theano.function(inputs = [a], outputs = [c])
output = f([[1, 2, 3, 4]])
```

(example taken from stackoverflow's forum, link)

```
#--------------------------------------------------------------------------
# Parameters to train
#--------------------------------------------------------------------------
trainables = []
if Win is not None:
    trainables += [Win]
trainables += [Wrec]
if Wout is not None:
    trainables += [Wout]
```

making a trainables list and adding input and output weight in it

```
if self.p['train_brec']:
    settings['train recurrent bias'] = 'yes'
    trainables += [brec]
else:
    settings['train recurrent bias'] = 'no'
```

if recurrent biases are to be train depending on the answer either add them to trainables list and put into settings dict that yes recurrent biases are being trained else just add no in settings dict for recurrent bias training

```
        if self.p['train_bout']:
            settings['train output bias'] = 'yes'
            trainables += [bout]
        else:
            settings['train output bias'] = 'no'
```

same logic as before

Note for our case both recurrent and output biases are not trainables

```
        # In continuous mode it doesn't make sense to train x0, which is
forgotten
        if self.p['mode'] == 'continuous':
            self.p['train_x0'] = False
```

if mode is batch don’t train x0 as it will be forgotten

For our case mode of training is batch(will be specified in upcoming code)

```
        if self.p['train_x0']:
            settings['train initial conditions'] = 'yes'
            trainables += [x0]
        else:
            settings['train initial conditions'] = 'no'
```

if x0 is to be trained or not is being decided here. For our case we choose to0 train x0

```
        #--------------------------------------------------------------------------
        # Weight matrices
        #--------------------------------------------------------------------------

        # Input
        if Nin > 0:
            if self.p['Cin'] is not None:
                C = self.p['Cin']
                settings['sparseness (Win)'] = ('p = {:.2f}, p_plastic = {:.2f}'
                                                .format(C.p, C.p_plastic))

                Cin_mask_plastic = theanotools.shared(C.mask_plastic)
                Cin_mask_fixed   = theanotools.shared(C.mask_fixed)

                Win_ = Cin_mask_plastic*Win + Cin_mask_fixed
                Win_.name = 'Win_'
            else:
                Win_ = Win
```

Now we make changes weight matrix if both, input is to be given to model is present along with connectivity mask

If they are then, add sparseness of weight matrix to settings dict along with probability of plastic values. Then make shared variable out of plastic and fixed values of input connectivity matrix

Finally sums up whole input weight matrix with joining of plastic and fixed mask to the equation and get our completed Win and then name it ‘Win_’(Note: this Win is now essentially a shared variable made from component shared variables Cin_mask_plastic and Cin_mask_fixed)

Otherwise just keep Win_ as simply Win.

```
        # Recurrent
        if self.p['Crec'] is not None:
            C = self.p['Crec']
            settings['sparseness (Wrec)'] = ('p = {:.2f}, p_plastic = {:.2f}'
                                             .format(C.p, C.p_plastic))

            Crec_mask_plastic = theanotools.shared(C.mask_plastic)
            Crec_mask_fixed   = theanotools.shared(C.mask_fixed)

            Wrec_ = Crec_mask_plastic*Wrec + Crec_mask_fixed
            Wrec_.name = 'Wrec_'
        else:
            Wrec_ = Wrec
```

```
        # Output
        if self.p['Cout'] is not None:
            C = self.p['Cout']
            settings['sparseness (Wout)'] = ('p = {:.2f}, p_plastic = {:.2f}'
                                             .format(C.p, C.p_plastic))

            Cout_mask_plastic = theanotools.shared(C.mask_plastic)
            Cout_mask_fixed   = theanotools.shared(C.mask_fixed)

            Wout_ = Cout_mask_plastic*Wout + Cout_mask_fixed
            Wout_.name = 'Wout_'
        else:
            Wout_ = Wout
```

Similar procedure as before output and recurrent weight matrices are created as well with labels ‘Wout_’ and ‘Wrec_’.

```
        #------------------------------------------------------------------------
        # Dale's law
        #------------------------------------------------------------------------
```

Section that governs with conversion of ei units to positive ones refer to paper to know why we are doing this

```
        if self.p['ei'] is not None:
            # Function to keep matrix elements positive
            if self.p['ei_positive_func'] == 'abs':
                settings['E/I positivity function'] = 'absolute value'
                make_positive = abs
            elif self.p['ei_positive_func'] == 'rectify':
                settings['E/I positivity function'] = 'rectify'
                make_positive = theanotools.rectify
            else:
                raise ValueError("Unknown ei_positive_func.")
```

conforming positive function we will use for ei units to make them all excitatory it can be absolute function or rectify function depending on whichever setting dict will be updated such and ‘make_positive’ variable will be assigned that function (abs() function takes absolute value of number and Theano.rectify function takes maximum number between, the num provided and 0 i.e., will take 0 for negative values)

```
        # Assume inputs are excitatory
        if Nin > 0:
            Win_ = make_positive(Win_)

        # E/I
        ei    = theanotools.shared(self.p['ei'], name='ei')
        Wrec_ = make_positive(Wrec_)*ei

        Wout_ = make_positive(Wout_)*ei
```

Assuming all input units as excitatory and making passing them through positive function

Next converting ei array to shared variable of Theano with name ‘ei’

Converting first all Wrec_ and Wout_ through positive function then mapping ei units by multiplying with ei array(now a shared variable) giving new shared variables Wrec_ and Wout_

```
        #--------------------------------------------------------------------
        # Variables to save
        #--------------------------------------------------------------------

        if Nin > 0:
            save_values = [Win_]
        else:
            save_values = [None]
        save_values += [Wrec_, Wout_, brec, bout, x0]
```

if number inputs are greater than zero then that means we will have to save input values so we will make new list that will have Win_ as its element

otherwise, we have no need to save input weight matrix

Regardless of above add Wrec_, Wout_, brec, bout and x0 to save_values list

```
        #--------------------------------------------------------------------
        # Activation functions
        #--------------------------------------------------------------------
```

Defining activation function to be used in training. Know activation function here

```
        f_hidden, d_f_hidden =
theanotools.hidden_activations[self.p['hidden_activation']]
        settings['hidden activation'] = self.p['hidden_activation']

        act = self.p['output_activation']
        f_output = theanotools.output_activations[act]
```

calling theanotools 'hidden activation' dictionary to get f_hidden and d_f_hidden (derivative of that hidden function)with key 'self.p['hidden_activation']' this key is name of activation function to be used (by default it is 'rectify') from which it will fetch that function from hidden_activation dictionary.

A look at hidden activation dictionary situated in theanotools

```
hidden_activations = {
    'linear':        (lambda x: x,   lambda x: 1),
    'rectify':       (rectify,       d_rectify),
    'rectify_power': (rectify_power, d_rectify_power),
    'sigmoid':       (sigmoid,       d_sigmoid),
    'tanh':          (tanh,          d_tanh),
    'rtanh':         (rtanh,         d_rtanh),
    'softplus':      (softplus,      d_softplus)
}
```

The functions are in tuple form and every tuple has 2 functions

Linear:

1st is simple and sweet linear function

Defined as to give back what it is given as it is

2nd is a function that gives back 1 for every x given to it i.e., its derivative will be used in sgd

Lambda in simple terms is ready to go function definer that can take many arguments and return their calculation over one expression. Read here about lambda in python

Rectify:

1st is Rectify function present in same file

```
if hasattr(T.nnet, 'relu'):
    rectify = T.nnet.relu
else:
    def rectify(x):
        return T.switch(x > 0, x, 0)
```

T denote theano.Tensor

```
import theano
import theano.tensor as T
```

if ReLU is available in Theano then use it as rectify function other wise it will use Theano.Tensor's switch method to define Relu function which takes max(0,x) for a given x

think of T.switch() as just another operator that acts on three symbolic variables, if the first is true, return the second, else return the third.

Next the 2nd element of tuples is it's derivative function

```
def d_rectify(x):
    return T.switch(x > 0, 1, 0)
```

just replace x by 1 in original function

rectify_power:

same as above logic

```
def rectify_power(x, n=2):
    return T.switch(x > 0, x**n, 0)

def d_rectify_power(x, n=2):
    return T.switch(x > 0, n*x**(n-1), 0)
```

if you understood the rectify then it is almost same just different 2nd argument

Sigmoid:

```
sigmoid = T.nnet.sigmoid

def d_sigmoid(x):
    return sigmoid(x)*(1 - sigmoid(x))
```

defining sigmoid variable as sigmoid function from T.nnet (nnetstands for neural network and possess various functions for neural network)

then just defines a derivative function for sigmoid. Nothing much to explain here check out sigmoid function on google and you are good to go

tanh:

```
tanh = T.tanh

def d_tanh(x):
    return 1 - tanh(x)**2
```

nothing new to explain here

rtanh:

```
def rtanh(x):
    return rectify(tanh(x))

def d_rtanh(x):
    return T.switch(x > 0, d_tanh(x), 0)
```

rectified tanh

nothing new to explain here

softplus:

```
def softplus(x):
    return T.log(1 + T.exp(x))

d_softplus = sigmoid
```

T.exp() calculates exponential

T.log calculates log

Alas these were all the functions inside the hidden activation functions dictionary and for our case we use rectify function

So, f_hidden is rectify function and d_f_hidden is derivative of rectify function

Back to trainer.py and moving to next line

```
        act = self.p['output_activation']
        f_output = theanotools.output_activations[act]
```

act is output activation function we get its value from self.p

f_output get its value from value at key the output function we will be using, inside dictionary 'output_activations'

Going to theanotools.py to see this dict

```
output_activations = {
    'linear':        (lambda x: x),
    'rectify':       rectify,
    'rectify_power': rectify_power,
    'sigmoid':       sigmoid,
    'softmax':       softmax
    }
```

Most function we can use are ones we already saw for hidden activation function just softmax is new here

Getting its definition

```
def softmax(x):
    """
    Softmax function.

    Parameters
    ----------

    x : theano.tensor.tensor3
        This function assumes the outputs are the third dimension of x.

    """
    sh = x.shape
    x  = x.reshape((sh[0]*sh[1], sh[2]))
    fx = T.nnet.softmax(x)
    fx = fx.reshape(sh)

    return fx
```

this function will take an 3 dimensional tensor x and take out its shape reshape it to 2d with shown size, apply softmax function and finally reshape it to original shape and return it

traveling back to trainer.py

In our case we had output activation function as linear

```
        if act == 'sigmoid':
            settings['output activation/loss'] = 'sigmoid/binary cross entropy'
            f_loss = theanotools.binary_crossentropy
        elif act == 'softmax':
            settings['output activation/loss'] = 'softmax/categorical cross
entropy'
            f_loss = theanotools.categorical_crossentropy
        else:
            settings['output activation/loss'] = act + '/squared'
            f_loss = theanotools.L2
```

here we define loss function depending on what output activation function we will be using and then introduce activation and loss function we are using in setting dictionary.

Note: for our case we are using linear activation function with squared loss

See here Theanotools definition of

binary_cross_entropy:

```
epsilon = 1e-10

def binary_crossentropy(y, t):
    return -t*T.log(y + epsilon) - (1-t)*T.log((1-y) + epsilon)
```

Nothing new to explain here just calculation from binary crossentropy formula and function return

categorical_crossentropy:

```
def categorical_crossentropy(y, t):
    return -t*T.log(y + epsilon)
```

Same here too

L2:

```
def L2(y, t):
    return (y - t)**2
```

Same here three

Back to trainer.py

```
        #--------------------------------------------------------------------
        # RNN
        #--------------------------------------------------------------------

        # Dims: time, trials, units
        # u[:,:,:Nin]  contains the inputs (including baseline and noise),
        # u[:,:,Nin:]  contains the recurrent noise
```

Defining RNN stuffs here

```
        u   = T.tensor3('u')
        x0_ = T.alloc(x0, u.shape[1], x0.shape[0])
```

creating a 3 – dimensional **symbolic tensor** variable

see here how tensors are in Theano and see here about symbolic tensors

next x0 is made a tensor with its same values but shape as specified by next 2 arguments

x0 is also a **symbolic tensor** now.

see .alloc()

theano.tensor.alloc(*value*, **shape*)[source]

Parameters:
- **value** – a value with which to fill the output
- **shape** – the dimensions of the returned array

Returns: an N-dimensional tensor initialized by *value* and having the specified shape.

Shape is specified as:

```
u shape:  Subtensor{int64}.0
x0 shape:  Subtensor{int64}.0
```

```
        if Nin > 0:
            def rnn(u_t, x_tm1, r_tm1, WinT, WrecT):
                x_t = ((1 - alpha)*x_tm1
                       + alpha*(T.dot(r_tm1, WrecT)        # Recurrent
```

```
                            + brec                      # Bias
                            + T.dot(u_t[:,:Nin], WinT) # Input
                            + u_t[:,Nin:])             # Recurrent noise
                    )
            r_t = f_hidden(x_t)

            return [x_t, r_t]
```

if Nin is not zero then define a function rnn that creates x_t and r_t as specified by code

recall alpha we assigned at init of trainer

here

```
        N     = self.p['N']
        Nin   = self.p['Nin']
        Nout  = self.p['Nout']
        alpha = self.p['dt']/self.p['tau']
```

other stuffs are just function's own arguments in the equation

T.dot() does dot product

u_t[: , : Nin] is taking every row of u_t but taking columns only till 'Nin' index

then pass this u_t to hidden activation function to get r_t as output

```
        [x, r], _ = theano.scan(fn=rnn,
                                outputs_info=[x0_, f_hidden(x0_)],
                                sequences=u,
                                non_sequences=[Win_.T, Wrec_.T])
```

next is .scan() function of Theano

It is very similar to for loop in python, the sequences argument is a iteratable item that it will iterate and perform operation given in 'fn'(rnn for our case) with function parameters taking as, 'sequences' followed by every value in 'output_info' and then every value in 'non_sequences'

Then after one iteration of the 'fn', return from it will be new output_info and then supplied back for next iteration similar to how initial out_info was supplied this process will go on updating output_info til sequences run out of values.

Scan results a tuples containing our result([x,r]) and dictionary of updates (which we ignore and do not use for any case) Note that the result is not a matrix, but a 3D tensor containing the return value from 'fn'(rnn) for each step

What scan does in summary has been explained above

See below more detailed explanation about scan for implementing in matlab

scan(fn, sequences=None, outputs_info=None, non_sequences=None, n_steps=None, truncate_gradient=-1, go_backwards=False, mode=None, name=None, profile=False, allow_gc=None, strict=False, return_list=False)

This function constructs and applies a Scan op to the provided arguments.

Parameters

---

fn

``fn`` is a function that describes the operations involved in one step of ``scan``. ``fn`` should construct variables describing the output of one iteration step. It should expect as input theano variables representing all the slices of the input sequences and previous values of the outputs, as well as all other arguments given to scan as ``non_sequences``. The order in which scan passes these variables to ``fn`` is the following :

* all time slices of the first sequence
* all time slices of the second sequence
* ...
* all time slices of the last sequence
* all past slices of the first output
* all past slices of the second output
* ...
* all past slices of the last output
* all other arguments (the list given as `non_sequences` to scan)

The order of the sequences is the same as the one in the list

`sequences` given to scan. The order of the outputs is the same

as the order of ``outputs_info``. For any sequence or output the

order of the time slices is the same as the one in which they have

been given as taps. For example if one writes the following :

```
 scan(fn, sequences = [ dict(input= Sequence1, taps = [-3,2,-1]), Sequence2

 , dict(input =  Sequence3, taps = 3) ], outputs_info = [ dict(initial
=  Output1, taps = [-3,-5]), dict(initial = Output2, taps = None), Output3 ]
                  , non_sequences = [ Argument1, Argument2])
```

``fn`` should expect the following arguments in this given order:

#. ``Sequence1[t-3]``

#. ``Sequence1[t+2]``

#. ``Sequence1[t-1]``

#. ``Sequence2[t]``

#. ``Sequence3[t+3]``

#. ``Output1[t-3]``

#. ``Output1[t-5]``

#. ``Output3[t-1]``

#. ``Argument1``

#. ``Argument2``

The list of ``non_sequences`` can also contain shared variables used in the function, though ``scan`` is able to figure those out on its own so they can be skipped. To some extend ``scan`` can also figure out other ``non sequences`` (not shared) even if not passed to scan (but used by `fn`). But to keep things neat we provide scan with 'non_sequences'

A simple example of this would be :

```
              import theano.tensor as TT

            W   = TT.matrix()
```

```
            W_2 = W**2
            def f(x):
                return TT.dot(x,W_2)
```

The function is expected to return two things. One is a list of outputs ordered in the same order as ``outputs_info``, with the difference that there should be only one output variable per output initial state (even if no tap value is used). Secondly `fn` should return an update dictionary (that tells how to update any shared variable after each iteration step). The dictionary can optionally be given as a list of tuples.

EXTRA about scan:{

To use ``scan`` as a while loop, the user needs to change the function ``fn`` such that also a stopping condition is returned. To do so, he/she needs to wrap the condition in an ``until`` class. The condition should be returned as a third element, for example:

```
            return [y1_t, y2_t], {x:x+1}, theano.scan_module.until(x < 50)
```

Note that a number of steps (considered in here as the maximum number of steps) is still required even though a condition is passed (and it is used to allocate memory if needed). = {}):

}

sequences

``sequences`` is the list of Theano variables or dictionaries describing the sequences ``scan`` has to iterate over.

Extra{

If a sequence is given as wrapped in a dictionary, then a set of optional information can be provided about the sequence. The dictionary

should have the following keys:

* ``input`` (*mandatory*) -- Theano variable representing the
  sequence.

* ``taps`` -- Temporal taps of the sequence required by ``fn``.
  They are provided as a list of integers, where a value ``k``
  impiles that at iteration step ``t`` scan will pass to ``fn``
  the slice ``t+k``. Default value is ``[0]``

Any Theano variable in the list ``sequences`` is automatically
wrapped into a dictionary where ``taps`` is set to ``[0]``
}

outputs_info
``outputs_info`` is the list of Theano variables or dictionaries
describing the initial state of the outputs computed
recurrently.
non_sequences
``non_sequences`` is the list of arguments that are passed to
``fn`` at each steps. One can opt to exclude variable
used in ``fn`` from this list as long as they are part of the
computational graph, though for clarity we encourage not to do so.

Extra unused parameter{
n_steps
``n_steps`` is the number of steps to iterate given as an int
or Theano scalar. If any of the input sequences do not have
enough elements, scan will raise an error. If the *value is 0* the

outputs will have *0 rows*. If n_steps is not provided, ``scan`` will

figure out the amount of steps it should run given its input

sequences. ``n_steps` < 0 is not supported anymore.

}

Returns

----

tuple

Tuple of the form (outputs, updates); ``outputs`` is either a

Theano variable or a list of Theano variables representing the

outputs of ``scan`` (in the same order as in ``outputs_info``).

``updates`` is a subclass of dictionary specifying the update rules for

all shared variables used in scan.

This dictionary should be passed to ``theano.function`` when you compile

your function. The change compared to a normal dictionary is that we

validate that keys are SharedVariable and addition of those dictionary

are validated to be consistent

See here to grasp a better idea of it through more examples

If you understand now what scan did then it's the end of 'if' statement and what we got is 'x' containing in it, the tensor with rnn function applied to all its element and 'r' that is hidden activation function applied to this 'x'.

Now the else statement will be exactly same but just without any trace of Win and Nin components

i.e., mapping the condition when Nin is not given or zero

Note: here a "_" can be seen while getting return from scan this is to skip storing 'updates' that come in result with 'values' (x,r).

```
        else:
            def rnn(u_t, x_tm1, r_tm1, WrecT):
                x_t = ((1 - alpha)*x_tm1
                       + alpha*(T.dot(r_tm1, WrecT) # Recurrent
```

```
                                + brec              # Bias
                                + u_t[:,Nin:]) # Recurrent noise
                        )
                r_t = f_hidden(x_t)

                return [x_t, r_t]

            [x, r], _ = theano.scan(fn=rnn,
                                    outputs_info=[x0_, f_hidden(x0_)],
                                    sequences=u,
                                    non_sequences=[Wrec_.T])
```

x and r :

```
x,r is  Subtensor{int64::}.0   Subtensor{int64::}.0
```

```
        #-----------------------------------------------------------------------
        # Running mode
        #-----------------------------------------------------------------------

        if self.p['mode'] == 'continuous':
            settings['mode'] = 'continuous'

            if self.p['n_gradient'] != 1:
                print("[ Trainer.train ] In continuous mode,"
                      " so we're setting n_gradient to 1.")
                self.p['n_gradient'] = 1

            x0_ = x[-1]
        else:
            settings['mode'] = 'batch'
```

Running mode declaration whether it will be ‘continuous’ or ‘batch’

If continuous

- set ‘n_gradient’ to 1
- New variable x0_, remember x contain updates from each step of function ‘rnn’ application so by specifying ‘x[-1]’ we are taking the final value taken by the output_info’s x and storing it in x0_

Else

Nothing special just set the mode in setting dictionary to batch

```
x0_ is :  Subtensor{int64}.0
```

```
        #--------------------------------------------------------------------------
        # Readout
        #--------------------------------------------------------------------------

        z = f_output(T.dot(r, Wout_.T) + bout)
```

recall:

- f_output is output activation function
- r was created from rnn function over page 78-83
- T.dot() is a dot product function inbuilt in Theano

---

theano.tensor.dot(*X*, *Y*)[source]

For 2-D arrays it is equivalent to matrix multiplication, and for 1-D arrays to inner product of vectors (without complex conjugation). For N dimensions it is a sum product over the last axis of a and the second-to-last of b:

Parameters:

- **X** (*symbolic tensor*) – left term
- **Y** (*symbolic tensor*) – right term

Return type: *symbolic matrix or vector*

Returns: the inner product of *X* and *Y*.

- Wout_ is output weight matrix we created over pages 70-71 and .T means taking transpose (Do **not** confuse this '.T' with 'T' for tensor module from Theano that we are using to create tensor)
- bout is output biases created over page 66.

End result we get **symbolic tensor** variable z that applies output activation function over output bias plus dot product of 'r' and output weight's transpose.

```
        #---------------------------------------------------------------------
        # Deduce whether the task specification contains an output mask -- use a
        # Temporary dataset so it doesn't affect the training.
        #---------------------------------------------------------------------

        dataset = Dataset(1, task, self.floatX, self.p, name='gradient')
        if dataset.has_output_mask():
            settings['output mask'] = 'yes'
        else:
            settings['output mask'] = 'no'
```

as mentioned in the comment this is a check for output mask where we use a temporary dataset for verify this

we are first using 'Dataset' file of pycog and making a temporary dataset here using Dataset class of prespecified module

Moving over to the class definition

```
class Dataset:
    """
    Dataset for training.
    """
    @staticmethod
    def rectify(x):
        return x*(x > 0)

    #////////////////////////////////////////////////////////////////////////////
```

We trigger the init function whenever we call upon our class by name with parameters

```
    def __init__(self, size, task, floatX, p, batch_size=None, seed=1,
name='Dataset'):
        """
```

```
        Parameters
        ----------

        size : int
               Number of trials in each minibatch.

        task : Python function

        floatX : dtype
                 Floating-point type for NumPy arrays.

        p : dict
            Parameters.

        batch_size : int, optional
                     Number of trials to store. If `None`, same as `size`.

        seed : int
               Seed for random number generator.

        name : str
               Name of the dataset, which can be used by `task`, e.g., to
distinguish
               between gradient and validation datasets.

        """
```

If you match the parameters we provided to class Dataset in trainer.py with parameters present here in init we notice:

- size is 1
- task is task variable that we are carrying over from model.py, that contains our generate trial function from S1_code.py. (see pages 29 for definition)
- floatx is self.floatx (**Note:** self here is of trainer class) see trainer.py init definition to see floatx origin.
  Floatx is specification of type of data like int64, float32 etc.
- p is self.p i.e., params
- batch_size and seed are skipped hence they will get default values
- name is set to gradient.

Moving on

```
        self.minibatch_size = size
        self.task           = task
        self.floatX         = floatX
        self.name           = name
        self.network_shape  = (p['Nin'], p['N'], p['Nout'])
```

assigniment of class variables

```
        for k in ['dt', 'rectify_inputs', 'baseline_in']:
            setattr(self, k, p[k])
```

a loop that adds 'dt', 'rectify_inputs' and 'baseline_in' as class variables , with corresponding values from ‘p’(params)

see info abt setattr() here

```
        # Rescale noise
        self.var_in  = 2*p['tau_in']/p['dt']*p['var_in']
        self.tau     = p['tau']
        self.var_rec = 2/p['dt']*p['var_rec']
```

more class variable assignments

```
        # Random number generator
        self.rng = np.random.RandomState(seed)
```

random number generator created from numpy random.RandomState with specified seed

see here np.random.RandomState

```
        # Batch size
        if batch_size is None or batch_size < size:
            batch_size = size
        self.batch_size = batch_size
```

if batch_size is None or either is less than size i.e., number of trials in each minibatch then set batch_size as size.

Add a new class variable batch_size with value batch_size

```
        # Initialize
        self.inputs    = None
        self.targets   = None
        self.trials    = []
        self.ntrials   = 0
        self.trial_idx = self.batch_size
```

more class variables assignments

This marks the end of init run of dataset.py and we are lead back to trainer.py's execution

```
if dataset.has_output_mask():
            settings['output mask'] = 'yes'
        else:
            settings['output mask'] = 'no'
```

next we call has_output_mask function of dataset class

so once again we are redirected to dataset.py

```
    #////////////////////////////////////////////////////////////////////////
    def has_output_mask(self):
        """
        Determine whether the outputs have a time mask.

        """
        # Generate a trial
        params = {
            'callback_results': None,
            'target_output':    True,
            'name':             self.name
            }
        trial = self.task.generate_trial(self.rng, self.dt, params)

        return ('mask' in trial)
```

This function will check whether outputs have time mask

To do this it will use generate trial function we created in S1_code.py supply it with dummy params as shown and generate a dummy trial

Then return the value of 'mask' key in the newly available 'trial' dictionary

Below is trial dictionary generated by has_output_mask function

```
Dataset.py dummy trial is  {'t': array([  20.,   40.,   60.,   80.,  100.,  120.,  140.,  160.,  180.,
       200.,  220.,  240.,  260.,  280.,  300.,  320.,  340.,  360.,
       380.,  400.,  420.,  440.,  460.,  480.,  500.,  520.,  540.,
       560.,  580.,  600.,  620.,  640.,  660.,  680.,  700.,  720.,
```

```
        740.,  760.,  780.,  800.,  820.,  840.,  860.,  880.,  900.,
        920.,  940.,  960.,  980., 1000., 1020., 1040., 1060., 1080.,
       1100., 1120., 1140., 1160., 1180., 1200.]), 'epochs': {'fixation': (0, 100), 'stimulus': (100,
900), 'decision': (900, 1200), 'T': 1200}, 'info': {'coh': 16, 'left_right': 1, 'choice': 0}, 'inputs':
array([[0.   , 0.   ],
       [0.   , 0.   ],
       [0.   , 0.   ],
       [0.   , 0.   ],
       [0.   , 0.   ],
       [0.756, 0.244],
       [0.756, 0.244],
       [0.756, 0.244],
       [0.756, 0.244],
       [0.756, 0.244],
       [0.756, 0.244],
       [0.756, 0.244],
       [0.756, 0.244],
       [0.756, 0.244],
       [0.756, 0.244],
       [0.756, 0.244],
       [0.756, 0.244],
       [0.756, 0.244],
       [0.756, 0.244],
       [0.756, 0.244],
       [0.756, 0.244],
       [0.756, 0.244],
       [0.756, 0.244],
       [0.756, 0.244],
       [0.756, 0.244],
       [0.756, 0.244],
       [0.756, 0.244],
       [0.756, 0.244],
       [0.756, 0.244],
       [0.756, 0.244],
       [0.756, 0.244],
       [0.756, 0.244],
       [0.756, 0.244],
       [0.756, 0.244],
       [0.756, 0.244],
       [0.756, 0.244],
       [0.756, 0.244],
       [0.756, 0.244],
       [0.756, 0.244],
       [0.756, 0.244],
```

```
       [0.756, 0.244],
       [0.756, 0.244],
       [0.756, 0.244],
       [0.756, 0.244],
       [0.   , 0.   ],
       [0.   , 0.   ],
       [0.   , 0.   ],
       [0.   , 0.   ],
       [0.   , 0.   ],
       [0.   , 0.   ],
       [0.   , 0.   ],
       [0.   , 0.   ],
       [0.   , 0.   ],
       [0.   , 0.   ],
       [0.   , 0.   ],
       [0.   , 0.   ],
       [0.   , 0.   ],
       [0.   , 0.   ],
       [0.   , 0.   ]]), 'outputs': array([[0.2, 0.2],
       [0.2, 0.2],
       [0.2, 0.2],
       [0.2, 0.2],
       [0.2, 0.2],
       [0. , 0. ],
       [0. , 0. ],
       [0. , 0. ],
       [0. , 0. ],
       [0. , 0. ],
       [0. , 0. ],
       [0. , 0. ],
       [0. , 0. ],
       [0. , 0. ],
       [0. , 0. ],
       [0. , 0. ],
       [0. , 0. ],
       [0. , 0. ],
       [0. , 0. ],
       [0. , 0. ],
       [0. , 0. ],
       [0. , 0. ],
       [0. , 0. ],
       [0. , 0. ],
       [0. , 0. ],
       [0. , 0. ],
       [0. , 0. ],
```

```
       [0. , 0. ],
       [0. , 0. ],
       [0. , 0. ],
       [0. , 0. ],
       [0. , 0. ],
       [0. , 0. ],
       [0. , 0. ],
       [0. , 0. ],
       [0. , 0. ],
       [0. , 0. ],
       [0. , 0. ],
       [0. , 0. ],
       [0. , 0. ],
       [0. , 0. ],
       [0. , 0. ],
       [0. , 0. ],
       [0. , 0. ],
       [0. , 0. ],
       [0. , 0. ],
       [0. , 0. ],
       [1. , 0.2],
       [1. , 0.2],
       [1. , 0.2],
       [1. , 0.2],
       [1. , 0.2],
       [1. , 0.2],
       [1. , 0.2],
       [1. , 0.2],
       [1. , 0.2],
       [1. , 0.2],
       [1. , 0.2],
       [1. , 0.2],
       [1. , 0.2],
       [1. , 0.2]]), 'mask': array([[1., 1.],
       [1., 1.],
       [1., 1.],
       [1., 1.],
       [1., 1.],
       [0., 0.],
       [0., 0.],
       [0., 0.],
       [0., 0.],
       [0., 0.],
       [0., 0.],
       [0., 0.],
       [0., 0.],
       [0., 0.],
```

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

```
    [1., 1.],
    [1., 1.]])}
```

So, what we will get as return will be mask array from this dict

Moving back to trainer.py

```
        if dataset.has_output_mask():
            settings['output mask'] = 'yes'
        else:
            settings['output mask'] = 'no'
```

here unless returned 'mask' is empty or None, the if condition will hold true otherwise else.

Depending on either 'settings' will be updated.

NOTE: - for S1_code case this will be always true as output mask is always generated, this condition is for in case some generate trial function of model files don't use output mask.

(Special note to ma'am: this function was the source of the params that we got from generate trial of S1_code when I was initially exploring stuffs and you enquired about the origin of that params)

Moving on

```
        #---------------------------------------------------------------------
        # Loss
        #---------------------------------------------------------------------
```

This section of trainer.py will define loss, error and regularization terms of training

```
        # (time, trials, outputs)
        target = T.tensor3('target')
```

new 3-D **symbolic tensor** 'target' is created

```
        # Set mask
        mask     = target[:,:,Nout:]
        masknorm = T.sum(mask)
```

set mask as third axis of target starting from 'Nout' to the end of axis.

Norm of mask defined right below is sum of mask (mask is tensor so T.sum() was used)

```
        # Input-output pairs
        inputs = [u, target]
        # target[:,:,:Nout] contains the target outputs, &
        # target[:,:,Nout:] contains the mask.
```

New variable inputs that store u: input symbolic tensor and target: output symbolic tensor in form of array

As specified in comments the third axis has as specified mask and target output

```
        # Loss, not including the regularization terms
        loss = T.sum(f_loss(z, target[:,:,:Nout])*mask)/masknorm
```

defining the loss as tensor sum of output from loss function multiplied by mask and the total is divided by ‘masknorm’

recall: ‘f_loss’ was defined along with output and input activation function around pages 76-77

```
        # Root-mean-squared error
        error = T.sqrt(T.sum(theanotools.L2(z, target[:,:,:Nout])*mask)/masknorm)
```

similar as above defining error

sqrt stands for square root

theanotools.L2 is L2 regularization function defined in theanotools.py

recall it was also a conditional candidate for ‘f_loss’

in such case it might be evident that error and loss are equivalent

theanotools.py definition of L2 function

```
def L2(y, t):
    return (y - t)**2
```

returns the squared difference

back to trainer.py

```
        #--------------------------------------------------------------------------
        # Regularization terms
        #--------------------------------------------------------------------------

        regs = 0

        #--------------------------------------------------------------------------
        # L1 weight regularization
        #--------------------------------------------------------------------------
```

```
        lambda1 = self.p['lambda1_in']
        if lambda1 > 0:
            settings['L1 weight regularization (Win)'] = ('lambda1_in = {}'
                                                          .format(lambda1))
            regs += lambda1 * T.mean(abs(Win))

        lambda1 = self.p['lambda1_rec']
        if lambda1 > 0:
            settings['L1 weight regularization (Wrec)'] = ('lambda1_rec = {}'
                                                           .format(lambda1))
            regs += lambda1 * T.mean(abs(Wrec))

        lambda1 = self.p['lambda1_out']
        if lambda1 > 0:
            settings['L1 weight regularization (Wout)'] = ('lambda1_out = {}'
                                                           .format(lambda1))
            regs += lambda1 * T.mean(abs(Wout))
```

'regs' is regularization term that will accumulate as we go

Starting with L1 regularization

Input Lambda is taken out of p dict (params) then check over condition if it is greater than zero(i.e., it is to be taken into consideration according to params) then update settings dict and aggregate it, with multiplication of lambda and input's mean after taking absolute, with regs

The same will repeat for output and recurrent Lambdas

End result we get is regs

A look at page 38 showing params dict will reveal that all the 3 condition will not held for S1_code hence regs is 0

```
Your regs is:  0
```

```
        #--------------------------------------------------------------------------
        # L2 weight regularization
        #--------------------------------------------------------------------------

        if Nin > 0:
            lambda2 = self.p['lambda2_in']
            if lambda2 > 0:
                settings['L2 weight regularization (Win)'] = ('lambda2_in = {}'
                                                              .format(lambda2))
```

```
                regs += lambda2 * T.mean(Win**2)

        lambda2 = self.p['lambda2_rec']
        if lambda2 > 0:
            settings['L2 weight regularization (Wrec)'] = ('lambda2_rec = {}'
                                                           .format(lambda2))
            regs += lambda2 * T.mean(Wrec**2)

        lambda2 = self.p['lambda2_out']
        if lambda2 > 0:
            settings['L2 weight regularization (Wout)'] = ('lambda2_out = {}'
                                                           .format(lambda2))
            regs += lambda2 * T.mean(Wout**2)
```

same as before just aggregating it with different formula

and again, same as before regs will be zero as all three ‘lambda_in’, ‘lambda_out’ and ‘lambda_rec’ are zero

```
Your regs is:  0
```

```
        #--------------------------------------------------------------------
        # L2 rate regularization
        #--------------------------------------------------------------------

        lambda2 = self.p['lambda2_r']
        if lambda2 > 0:
            settings['L2 rate regularization'] = 'lambda2_r = {}'.format(lambda2)
            regs += lambda2 * T.mean(r**2)
```

lambda2_r is also zero so this if condition will not hold hence keeping regs still zero

```
Your regs is:  0
```

Recall ‘r’ from Theano scan function in RNN section of trainer.py

```
        #--------------------------------------------------------------------------
        # Final costs
        #--------------------------------------------------------------------------

        costs = [loss, error]
```

costs array with loss and error

```
        #--------------------------------------------------------------------------
        # Datasets
        #--------------------------------------------------------------------------

        gradient_data   = Dataset(self.p['n_gradient'], task, self.floatX,
                                  self.p,
                                  batch_size=self.p['gradient_batch_size'],
                                  seed=self.p['gradient_seed'],
                                  name='gradient')
        validation_data = Dataset(self.p['n_validation'], task, self.floatX,
                                  self.p,
                                  batch_size=self.p['validation_batch_size'],
                                  seed=self.p['validation_seed'],
                                  name='validation')
```

these two will create an instance of class dataset with their as specified parameters

recall that we also used it when we checked if mask is present or not (‘has_output_mask’ function)

the execution for both will be same as then so if forgot then better brush up that, from page 85-88

moving on

```
        # Input noise
        if np.isscalar(self.p['var_in']):
            if Nin > 0:
                settings['sigma_in'] = '{}'.format(np.sqrt(self.p['var_in']))
        else:
            settings['sigma_in'] = 'array'
```

checks if var_in stands for variable input, is scalar or not

if yes check Nin >0 and if also yes then update settings with value as squareroot of var_in value

{} is place holder in python and .format(value) is used to specify the value for that place

If 1st condition does not hold update settings dictionary with string ‘array’ as value i.e., we are specifying sigma input will be an array

```
        # Recurrent noise
        if np.isscalar(self.p['var_rec']):
            settings['sigma_rec'] = '{}'.format(np.sqrt(self.p['var_rec']))
        else:
            settings['sigma_rec'] = 'array'
```

same logic as input noise

```
        # Dataset settings
        settings['rectify inputs']            = self.p['rectify_inputs']
        settings['gradient minibatch size']   = gradient_data.minibatch_size
        settings['validation minibatch size'] = validation_data.minibatch_size
```

more settings assignments

```
rectify inputs is:  True
gradient minibatch size: 20
validation minibatch size: 1100

        #---------------------------------------------------------------------
        # Other settings
        #---------------------------------------------------------------------

        settings['dt'] = '{} ms'.format(self.p['dt'])
        if np.isscalar(self.p['tau']):
            settings['tau'] = '{} ms'.format(self.p['tau'])
        else:
            settings['tau'] = 'custom'
        settings['tau_in']            = '{} ms'.format(self.p['tau_in'])
        settings['learning rate']     = '{}'.format(self.p['learning_rate'])
        settings['lambda_Omega']      = '{}'.format(self.p['lambda_Omega'])
        settings['max gradient norm'] = '{}'.format(self.p['max_gradient_norm'])
```

same logic ‘sigma_in’ and ‘sigma_rec’ for ‘tau’

other than that, all are new key-value additions in settings dict

```
        #--------------------------------------------------------------------
        # A few important Theano settings
        #--------------------------------------------------------------------

        settings['(Theano) floatX']   = self.floatX
        settings['(Theano) allow_gc'] = theano.config.allow_gc
```

Theano settings update

.floatX

String value: either ‘float64’ or ‘float32’

Default: ‘float64’

This sets the default dtype returned by tensor.matrix(), tensor.vector(), and similar functions. It also sets the default theano bit width for arguments passed as Python floating-point numbers.

.allow_gc

Bool value: either `True` or `False`

Default: `True`

This sets the default for the use of the Theano garbage collector for intermediate results. To use less memory, Theano frees the intermediate results as soon as they are no longer needed. Disabling Theano garbage collection allows Theano to reuse buffers for intermediate results between function calls. This speeds up Theano by no longer spending time reallocating space. This gives significant speed up on functions with many ops that are fast to execute, but this increases Theano’s memory usage.

```
(Theano) floatX:           float32
(Theano) allow_gc:         False
```

```
        #--------------------------------------------------------------------
        # Train!
        #--------------------------------------------------------------------

        print_settings(settings)
```

Time to start the training but before that we print out the settings

You see when we train model we get a long list of parameters settings those are from this

This print settings func was imported from utils.py

```
from .utils         import print_settings
```

moving to utils.py

```
def print_settings(settings, indent=3, title="=> settings"):
    """
    Pretty print.

    """
    print(title)
    maxlen = max([len(s) for s in settings])
    for k, v in settings.items():
        print(indent*' ' + '| {}:{}{}'.format(k, (maxlen - len(k) + 1)*' ', v))
    sys.stdout.flush()
```

it's a flasgy way of prining the settings used by our model

we print the title then get the maximum of length of items present in settings

then a for loop takes one key and its values at time and then print the info

indent*' ' given indentation i.e., spaces 'indent' times then some decorations and then according to length equally spaced values are printed

sys.stdout.flush() is a necessary function that ensures that print statement is not kept on hold i.e., not buffered up

for better explanation refer here

```
        sgd = SGD(trainables, inputs, costs, regs, x, z, self.p, save_values,
                  {'Wrec_': Wrec_, 'd_f_hidden': d_f_hidden})
        sgd.train(gradient_data, validation_data, savefile)
```

Now we finally start the real training

Up until now what we did, was all defining essential terms for training now we will get onto what training of the model by stochastic gradient descent method

First, we initialise the SGD class

To see its execution, we move over to sgd.py

```
class SGD:
    """
    Stochastic gradient descent training for RNNs.
```

```
    """

    @staticmethod
    def clip_norm(v, norm, maxnorm):
        """
        Renormalize the vector `v` if `norm` exceeds `maxnorm`.

        """
        return T.switch(norm > maxnorm, maxnorm*v/norm, v)
```

clip_norm function that does as the comments says

recall T.switch(), it is an operator that acts on three symbolic variables, if the first is true, return the second, else return the third.

```
    def __init__(self, trainables, inputs, costs, regs, x, z, params,
save_values,
                 extras):
        """
        Construct the necessary Theano functions.

        Parameters
        ----------

        trainables : list
                     List of Theano variables to optimize.

        inputs : [inputs, targets]
                 Dataset used to train the RNN.

        costs : [loss, ...]
                `costs[0]` is the loss that is optimized. `costs[1:]` are used
                for monitoring only.

        regs : Theano variable
               Regularization terms to add to costs[0].

        x : Theano variable
            Hidden unit activities.

        z : Theano variable
            Outputs.

        params : dict
                 All parameters associated with the training of this network --
```

```
this will be saved as part of the RNN savefile.

        save_values : list
                      List of Theano variables to save.

        extras : dict
                 Additinal information needed by the SGD training algorithm
                 (specifically, for computing the regularization term) that may
                 not
                 be needed by other training algorithms (e.g., Hessian-free).

        """
```

Take your time and read comments to get clarification about variables we are using

So, on inspection between our supplied parameters and init parameters we found:

- 'trainables' is 'trainables' from trainer.py, an array with values as, variables as to be updated during training

```
Trainables:  [Win, Wrec, Wout, x0]
```

- 'inputs' is 'inputs' array we created over page 94 that stores input-output pair

```
inputs is :  [u, target]
```

- 'costs' is 'costs' from trainer.py, we defined over page 97 containing loss and error

```
costs is :  [Elemwise{true_div,no_inplace}.0, Elemwise{sqrt,no_inplace}.0]
```

- 'regs' is as specified in comments

```
regs is :  0
```

- x is symbolic tensor for hidden units and z is output function made with symbolic tensors as placeholders

```
x is :  Subtensor{int64::}.0
z is :  Elemwise{add,no_inplace}.0
```

- params is self.p from trainer.py and it holds parameters that are needed for training

```
self.p is :  {'Nin': 2, 'N': 100, 'Nout': 2, 'Cout':
<pycog.connectivity.Connectivity object at 0x0000020C004F9F40>, 'ei': array([ 1,
1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,
       1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,
       1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,
       1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,
```

```
       1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1, -1, -1, -1, -1, -1,
      -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1]),
'n_validation': 1100, 'performance': <function performance_2afc at
0x0000020C004C79D0>, 'terminate': <function terminate at 0x0000020C004D7AF0>,
'extra_info': {}, 'rectify_inputs': True, 'train_brec': False, 'brec': 0,
'train_bout': False, 'bout': 0, 'train_x0': True, 'x0': 0.1, 'mode': 'batch',
'tau': 100, 'tau_in': 100, 'Cin': None, 'Crec': <pycog.connectivity.Connectivity
object at 0x0000020C00513100>, 'ei_positive_func': 'rectify',
'hidden_activation': 'rectify', 'output_activation': 'linear', 'n_gradient': 20,
'gradient_batch_size': None, 'validation_batch_size': None, 'lambda_Omega': 2,
'lambda1_in': 0, 'lambda1_rec': 0, 'lambda1_out': 0, 'lambda2_in': 0,
'lambda2_rec': 0, 'lambda2_out': 0, 'lambda2_r': 0, 'callback': None,
'min_error': 0, 'learning_rate': 0.01, 'max_gradient_norm': 1, 'bound': 1e-20,
'baseline_in': 0.2, 'var_in': 0.0001, 'var_rec': 0.0225, 'seed': 1234,
'gradient_seed': 11, 'validation_seed': 22, 'structure': {}, 'rho0': 1.5,
'max_iter': 10000000, 'dt': 20.0, 'distribution_in': 'uniform',
'distribution_rec': 'gamma', 'distribution_out': 'uniform', 'gamma_k': 2,
'checkfreq': None, 'patience': None, 'momentum': False, 'method': 'sgd'}
```

- ‘save_values’ is ‘save_values’ from trainer.py, list of Theano variables to be saved, see its definition over page 71 for better view of it than below

```
save_values is :  [Elemwise{mul,no_inplace}.0, Elemwise{mul,no_inplace}.0,
Elemwise{mul,no_inplace}.0, brec, bout, x0]
```

- Last is ‘extras’ that we have supplied as dictionary in trainer.py

```
{'Wrec_': Wrec_, 'd_f_hidden': d_f_hidden}
```

Wrec_ is recurrent weight matrix builded up from page 66 and 69 to 71

d_f_hidden is derivative of hidden activation function

Moving over to init’s code

```
        self.trainables  = trainables
        self.p           = params
        self.save_values = save_values

        # Trainable variable names
        self.trainable_names = [tr.name for tr in trainables]
```

Class variable definitions

The trainables variable names, class variable is defined so as to take in names of a tensor variables in trainables array and store them in an array

```
        #-----------------------------------------------------------------------
        # Setup
        #-----------------------------------------------------------------------

        lambda_Omega = T.scalar('lambda_Omega')
        lr           = T.scalar('lr')
        maxnorm      = T.scalar('maxnorm')
        bound        = T.scalar('bound')
```

Creating Theano Tensor scalars

'lr' is left-right

'maxnorm' is maximum gradient norm

```
        #-----------------------------------------------------------------------
        # Compute gradient
        #-----------------------------------------------------------------------

        # Pascanu's trick for getting dL/dxt
        # scan_node.op.n_seqs is the number of sequences in the scan
        # init_x is the initial value of x at all time points, including x0
        scan_node = x.owner.inputs[0].owner
        assert isinstance(scan_node.op, theano.scan_module.scan_op.Scan)
        npos   = scan_node.op.n_seqs + 1
        init_x = scan_node.inputs[npos]
        g_x,   = theanotools.grad(costs[0], [init_x])
```

The first line of comment is 'pascanu's trick for dL/dxt'

If you are wondering what it is means check out the top of the SGD.py where they have given reference about R. Pascan's research paper whom which many modifications are based on.

Link

If you move over to given GitHub link and check line 167 from RNN.py there, you will find the exact same 3 lines written right after the comments.

Scan_node:

Printing scan_node gives

```
Scan node: for{cpu,scan_fn}(Subtensor{int64}.0, Subtensor{:int64:}.0,
IncSubtensor{Set;:int64:}.0, IncSubtensor{Set;:int64:}.0, brec,
InplaceDimShuffle{1,0}.0, InplaceDimShuffle{1,0}.0)
```

n_pos and init_x does exactly what the comments says about them

‘x’ represents input

```
npos :  2
init_x :  IncSubtensor{Set;:int64:}.0
g_x :  Subtensor{::int64}.0
```

```
        # Get into "standard" order, by filling `self.trainables` with
        # `None`s if some of the parameters are not trained.
        Win, Wrec, Wout, brec, bout, x0 = RNN.fill(self.trainables,
self.trainable_names)
```

As comments say the RNN.fill() function will fill out ‘None’ in whichever supposed trainables are not to be trained

Definition of fill function in RNN.py:

```
    @staticmethod
    def fill(p_, varlist, all=['Win', 'Wrec', 'Wout', 'brec', 'bout', 'x0']):
        """
        Fill `p_` with `None`s if some of the parameters are not trained.

        """
        p = list(p_)
        for var in all:
```

```
            if var not in varlist:
                p.insert(all.index(var), None)

        return p
```

it takes trainables, their names and optionally names of all ‘supposed’ trainables as parameters (in case we want to modify some stuffs and require more or less parameters will be train)

- converts trainables to list, in case it’s not a array just for surety
- run a for loop for every value in ‘all’ (total trainable name possible)
- check if any variable name is absent or not if it is update ‘p’ by None at the position of that supposed trainable(p will have only trainables to be trained from trainer.py so it will always be <=’all’ in length but after going through this function it will be =’all’ for sure)
- return p

this returned p is an array and is split-stored in Win, Wrec, Wout, brec, bout and x0

```
        # Gradients
        g = theanotools.grad(costs[0] + regs, self.trainables)
        g_Win, g_Wrec, g_Wout, g_brec, g_bout, g_x0 = RNN.fill(g,
self.trainable_names)
```

the above function will call upon theanotools.grad() function which in turn will call upon Tensor.grad() function that will return gradient values calculated with respect to individual supplied list items

definition of grad function in theanotools.py

```
def grad(*args, **kwargs):
    kwargs.setdefault('disconnected_inputs', 'warn')

    return T.grad(*args, **kwargs)
```

the *args is used to send a non-keyworded variable-length argument list to functions.

While **kwargs will send variable length dictionary to function

For examples: We supplied two arguments to function so we get args and kwargs as

```
args = (Elemwise{true_div,no_inplace}.0, [IncSubtensor{Set;:int64:}.0])
kwargs = {}
```

the kwargs is empty because we didn’t specified it in key value pair

like if we had supplied grad.(some_arg=value)

then kwargs will be {some_arg:value}

next setdefault() if you recall we used it initially in S1_code py where it did the job of checking if specified first argument is present as key in dictionary or not in case it isn't then it will insert that key with specified value in second argument.

T.grad:

theano.gradient.grad(*cost, wrt, consider_constant=None, disconnected_inputs='raise', add_names=True, known_grads=None, return_disconnected='zero', null_gradients='raise'*)[source]

Return symbolic gradients of one cost with respect to one or more variables.

For more information about how automatic differentiation works in Theano, see `gradient`. For information on how to implement the gradient of a certain Op, see `grad()`.

Parameters:

- **cost** (`Variable` scalar (0-dimensional) tensor variable or `None`) – Value that we are differentiating (that we want the gradient of). May be *None* if *known_grads* is provided.
- **wrt** (`Variable` or list of Variables) – Term[s] with respect to which we want gradients
- **consider_constant** (*list of variables*) – Expressions not to backpropagate through
- **disconnected_inputs** (*{'ignore', 'warn', 'raise'}*) –

  Defines the behaviour if some of the variables in *wrt* are not part of the computational graph computing *cost* (or if all links are non-differentiable). The possible values are:
  - 'ignore': considers that the gradient on these parameters is zero.
  - 'warn': consider the gradient zero, and print a warning.
  - 'raise': raise DisconnectedInputError.
- **add_names** (*bool*) – If True, variables generated by grad will be named (d<cost.name>/d<wrt.name>) provided that both cost and wrt have names
- **known_grads** (*OrderedDict, optional*) – A ordered dictionary mapping variables to their gradients. This is useful in the case where you know the gradient on some variables but do not know the original cost.
- **return_disconnected** (*{'zero', 'None', 'Disconnected'}*) –
  - 'zero' : If wrt[i] is disconnected, return value i will be wrt[i].zeros_like()
  - 'None' : If wrt[i] is disconnected, return value i will be None

- ’Disconnected’ : returns variables of type DisconnectedType

- **null_gradients** ({*'raise', 'return'*}) –

  Defines the behaviour if some of the variables in *wrt* have a null gradient. The possibles values are:

  - ’raise’ : raise a NullTypeGradError exception
  - ’return’ : return the null gradients

Returns:	Symbolic expression of gradient of *cost* with respect to each of the *wrt* terms. If an element of *wrt* is not differentiable with respect to the output, then a zero variable is returned.
Return type:	variable or list/tuple of variables (matches *wrt*)

See the gradient page for complete documentation of the gradient module.

```
        #------------------------------------------------------------------------
        # Regularization for the vanishing gradient problem
        #------------------------------------------------------------------------

        if np.isscalar(self.p['tau']):
            alpha = T.scalar('alpha')
        else:
            alpha = T.vector('alpha')
        d_xt = T.tensor3('d_xt') # Later replaced by g_x, of size (time+1),
batchsize, N
        xt   = T.tensor3('xt')   # Later replaced by x, of size time, batchsize,
N
        # Using temporary variables instead of actual x variables
        # allows for calculation of immediate derivatives
```

Checks if tau is scalar

- if yes, creates alpha scalar variable
- otherwise, alpha vector variable

Creates 2 new 3-D symbolic tensor variables, d_xt and xt

Read comments carefully, it says g_x and x about being replaced head over to page 11 where we encounter the code where these values are replaced

```
        # Here construct the regularizer Omega for the vanishing gradient problem

        # Numerator of Omega (d_xt[1:] returns time X batchsize X N)
        # Notice Wrec_ is used in the network equation as: T.dot(r_tm1, Wrec_.T)
        num = (1 - alpha)*d_xt[1:] + T.dot(alpha*d_xt[1:],
self.Wrec_)*d_f_hidden(xt)
        num = (num**2).sum(axis=2)
```

operational statement self-explanatory stuffs

num will be symbolic denotation of specified operations over symbolic tensors

i.e., it will change throughout training as values inside of it changes

Again, Read comments carefully

```
# Denominator of Omega, small denominators are not considered
        # \partial E/\partial x_{t+1}, squared and summed over hidden units
        denom = (d_xt[1:]**2).sum(axis=2)
        Omega = (T.switch(T.ge(denom, bound), num/denom, 1) - 1)**2
```

Remember switch we used way back in this document if u forgot then check out page 72

T.ge is:

theano.tensor.ge(*a*, *b*)[source]

> Returns a variable representing the result of logical greater or equal than (a>=b).
>
> Also available using syntax `a >= b`

```
        # First averaged across batches (.mean(axis=1)),
        # then averaged across all time steps where |\p E/\p x_t|^2 > bound
        nelems = T.mean(T.ge(denom, bound), axis=1)
        Omega  = Omega.mean(axis=1).sum()/nelems.sum()
```

Same as comments say

```
        # tmp_g_Wrec: immediate derivative of Omega with respect to Wrec
        # Notice grad is computed before the clone.
        # This is critical for calculating the immediate derivative.
        tmp_g_Wrec = theanotools.grad(Omega, Wrec)
        Omega, tmp_g_Wrec, nelems = theano.clone([Omega, tmp_g_Wrec,
nelems.mean()],
                                                 replace=[(d_xt, g_x), (xt, x)])
        # Add the gradient to the original gradient
        g_Wrec += lambda_Omega * tmp_g_Wrec
```

Same as comments says

Just one thing to note that was hinted in comments on page 109 about replacing of d_xt and xt by g_x and x

What theano.clone does is basically it takes original list along with elements to be replaced in shown format and replace them in original list giving a cloned list with forementioned elements replaced from original list's element's definition

```
        #---------------------------------------------------------------------
        # Gradient clipping
        #---------------------------------------------------------------------

        g = []
        if 'Win' in self.trainable_names:
            g += [g_Win]
        g += [g_Wrec, g_Wout]
        if 'brec' in self.trainable_names:
            g += [g_brec]
        if 'bout' in self.trainable_names:
            g += [g_bout]
        if 'x0' in self.trainable_names:
            g += [g_x0]
```

- creating empty list 'g'
- according to what to train and what not to added gradients
- in case you wonder a clarification, those that were not to be trained will be 'None' here as done by rnn.fill() function

```
        # Clip
        gnorm = T.sqrt(sum([(i**2).sum() for i in g]))
        g = [SGD.clip_norm(i, gnorm, maxnorm) for i in g]
        g_Win, g_Wrec, g_Wout, g_brec, g_bout, g_x0 = RNN.fill(g,
self.trainable_names)
```

gnorm: gradient norm

([(i**2).sum() for i in g] : this line here will return a list with every element (i**2).sum() for every 'i' element present in 'g'

Then SGD class's 'clip_norm' function is called upon with parameters 'i', 'gnorm' and 'maxnorm' for every element 'i' present in 'g' i.e., clip_norm function will run g times with different i^{th} element from g every iteration

SGD.clip_norm() definition:

```
    @staticmethod
    def clip_norm(v, norm, maxnorm):
        """
        Renormalize the vector `v` if `norm` exceeds `maxnorm`.

        """
        return T.switch(norm > maxnorm, maxnorm*v/norm, v)
```

Already clarified over page 102

```
        # Pascanu's safeguard for numerical precision issues with float32
        new_cond = T.or_(T.or_(T.isnan(gnorm), T.isinf(gnorm)),
                         T.or_(gnorm < 0, gnorm > 1e10))
        if 'Win' in self.trainable_names:
            g_Win  = T.switch(new_cond, np.float32(0), g_Win)
        g_Wrec = T.switch(new_cond, np.float32(0.02)*Wrec, g_Wrec)
        g_Wout = T.switch(new_cond, np.float32(0), g_Wout)
        if 'brec' in self.trainable_names:
            g_brec = T.switch(new_cond, np.float32(0), g_brec)
        if 'bout' in self.trainable_names:
            g_bout = T.switch(new_cond, np.float32(0), g_bout)
        if 'x0' in self.trainable_names:
            g_x0 = T.switch(new_cond, np.float32(0), g_x0)
```

- Theano.tensor.or_()

theano.tensor.or_(*a*, *b*)[source]

Returns a variable representing the result of the bitwise or

- Theano.tensor.isnan()

theano.tensor.isnan(*a*)[source]

Returns a variable representing the comparison of `a` elements with nan.

This is equivalent to `numpy.isnan`.

- Theano.tensor.isinf()

theano.tensor.isinf(*a*)[source]

Returns a variable representing the comparison of `a` elements with inf or -inf.

This is equivalent to `numpy.isinf`.

- 1e10 represent 10^{10}

Inspecting these you can easily figure out the operation being done to create 'new_cond' variable(new condition variable)

Next up is bunch of if statements

As Wrec and Wout are mandatory there is no if trainable condition imposed over them, they go through the switch statement and work around if condition holds or not

Same will happen for Trainable items Win, b_rec etc.

```
        #--------------------------------------------------------------------------
        # Training step
        #--------------------------------------------------------------------------

        # Final gradients
        g = []
        if 'Win' in self.trainable_names:
            g += [g_Win]
        g += [g_Wrec, g_Wout]
        if 'brec' in self.trainable_names:
            g += [g_brec]
        if 'bout' in self.trainable_names:
            g += [g_bout]
        if 'x0' in self.trainable_names:
            g += [g_x0]
```

Same thing repeat that we did in gradient clipping start

```
        # Update rule
        updates = [(theta, theta - lr*grad) for theta, grad in
zip(self.trainables, g)]
```

above code is specifying the update rule

now if you understood a some time ago code where [(i**2).mean() for i in g] then zip() one is same as that just that there we were taking one element at a time but now we are taking theta and grad both simultaneously from respectively self.trainables and g, and zip() helps in this task

check here zip() function

```
        # Update function
        self.train_step = theanotools.function(
            inputs + [alpha, lambda_Omega, lr, maxnorm, bound],
            [costs[0] + regs, gnorm, Omega, nelems, x],
            updates=updates
            )
```

A new class variable definition ‘train_step’ created using theanotools.function() function

Theanotools.function class definition in theanotools

```
def function(*args, **kwargs):
    kwargs.setdefault('on_unused_input', 'warn')

    return theano.function(*args, **kwargs)
```

'on_unsused_input' argument specifies What to do if a variable in the 'inputs' list is not used in the graph. Possible values are 'raise', 'warn', and 'ignore'.

What theano.function() does is it takes definitive first argument as input, second argument as output and all afterwards are optional arguments including updates and then creates callable object over them from which the outputs are calculated provided the inputs using the already specified relations in between them.

```
        # Cost function
        self.f_cost = theanotools.function(inputs, [costs[0] + regs] + costs[1:]
+ [z])
```

same logic as before we create cost function

This right here ends the execution of init function of SGD.py we return back to trainer.py

The last line of trainer.py is:

```
        sgd.train(gradient_data, validation_data, savefile)
```

which once and for final calls upon the SGD.py

Train function:

```
    def train(self, gradient_data, validation_data, savefile):
        """
        Train the RNN.

        Paramters
        ---------

        gradient_data : pycog.Dataset
                        Gradient dataset.

        validation_data : pycog.Dataset
                          Validation dataset.

        savefile : str
                   File to save network information in.

        """
```

Read the comments carefully

On inspection the required parameters have same variable names as they have in trainer.py

```
        checkfreq = self.p['checkfreq']
        if checkfreq is None:
            checkfreq = int(1e4)//gradient_data.minibatch_size
```

get value of checkfreq key from params dictionary

if it is None give it value: 10^4//gradient_data.minibatch_size

recall gradient_data is Dataset class object and minibatch is one of its class's parameters

```
gradient_data is :  <pycog.dataset.Dataset object at 0x00000236F5CFE790>
validation_data is :  <pycog.dataset.Dataset object at 0x00000236F5D0B4F0>

minibatch :  20
checkfreq :  500
```

```
        patience = self.p['patience']
        if patience is None:
            patience = 100*checkfreq
```

get patience value from params same logic as before if none set as specified

patience value tells the trainer, how many epochs it must continue after the loss stopped from decreasing.

If you don't set patience value, the training will continue for all the epochs you set, even if your training results are not getting any better. This value saves you some time and energy

```
        alpha        = self.p['dt']/self.p['tau']
        lambda_Omega = self.p['lambda_Omega']
        lr           = self.p['learning_rate']
        maxnorm      = self.p['max_gradient_norm']
        bound        = self.p['bound']
        save_exclude = ['callback', 'performance', 'terminate']
```

giving class variables simpler names

and defining items excluded from save

```
        #---------------------------------------------------------------------
        # Continue previous run if we can
        #---------------------------------------------------------------------

        if os.path.isfile(savefile):
            with open(savefile,'rb') as f:
                save = pickle.load(f)

        if os.path.isfile(savefile):
            with open(savefile) as f:
                save = pickle.load(f)
            best          = save['best']
            init_p        = save['current']
            first_iter    = save['iter']
            costs_history = save['costs_history']
            Omega_history = save['Omega_history']

            # Restore RNGs for datasets
            gradient_data.rng   = save['rng_gradient']
            validation_data.rng = save['rng_validation']

            # Restore parameter values
            for i, j in zip(self.trainables, init_p):
                i.set_value(j)

            print(("[ {}.SGD.train ] Recovered saved model,"
```

```
                " continuing from iteration {}.").format(THIS, first_iter))
```

In we had a previous run of the model and we want to see it till the end then this function will help us out.

- os.path.isfile(file):

Syntax: *os.path.isfile(path)*

Parameter:
path*: A path-like object representing a file system path. A path-like object is either a string or bytes object representing a path.*

Return Type: *This method returns a Boolean value of class bool. This method returns True if specified path is an existing regular file, otherwise returns False.*

- With open(savefile) as f: i.e., opening savefile and naming its object 'f' temporarily
- Creating object and loading pickle file from path 'f'
- Take out relevant stuffs from the save
- Restore rng
- Restore parameters
- Print as specified

The save file will have previous runs data a sample print of of save file will look like

Otherwise if we do not have previous save_file

```
        else:
            best = {
                'iter':        1,
                'cost':        np.inf,
                'other_costs': [],
                'params':      SGD.get_values(self.save_values)
                }
            first_iter    = best['iter']
            costs_history = []
            Omega_history = []

            # Save initial conditions
            save = {
                'params':         {k: v for k, v in self.p.items()
                                   if k not in save_exclude},
                'varlist':        self.trainable_names,
                'iter':           1,
```

```
            'current':        SGD.get_values(self.trainables),
            'best':           best,
            'costs_history':  costs_history,
            'Omega_history':  Omega_history,
            'rng_gradient':   gradient_data.rng,
            'rng_validation': validation_data.rng
            }
        base, ext = os.path.splitext(savefile)
        dump(base + '_init' + ext, save)
```

- make dictionary ‘best’ that will store ‘iter’, ‘costs’ etc.
- make variable first iter with value from key iter in ‘best’ dictionary
- 2 new list for omega history and costs history
- Save initial conditions
- Create base file name and extension variable that get their value from os.path.splittext()

Syntax: *os.path.splitext(path)*

Parameter:
path*: A path-like object representing a file system path. A path-like object is either a str or bytes object representing a path.*

Return Type: *This method returns a tuple that represents root and ext(extension) part of the specified path name*

Carrying on

```
        #--------------------------------------------------------------------------
        # Updates
        #--------------------------------------------------------------------------

        performance = self.p['performance']
        terminate   = self.p['terminate']
        tr_Omega    = None
        tr_gnorm    = None
```

variables creation

```
        try:
            tstart = datetime.datetime.now()
            for iter in range(first_iter, 1+self.p['max_iter']):
                if iter % checkfreq == 1:
```

The Code under Try is, first exceuted if it runs into some error then it stops and exception statement takes over

The except case for our case is :

```
        except KeyboardInterrupt:
            print("[ {}.SGD.train ] Training interrupted by user during iteration
{}."
                  .format(THIS, iter))
```

This try and except is to map cases when user presses keys that drops any execution to error so for those cases the program shows the above print output rather than tons of frustrating red lines.

Focusing back to our try statement then

```
            tstart = datetime.datetime.now()
            for iter in range(first_iter, 1+self.p['max_iter']):
                if iter % checkfreq == 1:
```

record the start time

For every int ‘iter’ in range from ‘first_iter’ to self.p(“max_iter”)+1

(+1 because range stops at last value-1 of range)

- Check if iter%freq is 1 i.e., the remainder of them is 1:
    - In such case

```
                    #-------------------------------------------------------------
                    # Timestamp
                    #-------------------------------------------------------------

                    tnow      = datetime.datetime.now()
                    totalsecs = (tnow - tstart).total_seconds()

                    hrs  = int(totalsecs//3600)
                    mins = int(totalsecs%3600)//60
                    secs = int(totalsecs%60)

                    timestamp = tnow.strftime('%b %d %Y %I:%M:%S %p').replace('
0', ' ')
                    print('{} updates - {} ({} hrs {} mins {} secs elapsed)'
```

```
                    .format(iter-1, timestamp, hrs, mins, secs))
```

- Record current time
- Record difference in current and start time
- Conversion of difference in time we got to hrs, mins and seconds
- strftime(format)
  Returns : It returns the string representation of the date or time object

Directive	Meaning	Output Format
%b	Abbreviated month name.	Jan, Feb, …, Dec
%d	Day of the month as a zero added decimal.	01, 02, …, 31
%Y	Year with century as a decimal number.	2013, 2019 etc.
%I	Hour (12-hour clock) as a zero added decimal number.	01, 02, …, 12
%M	Minute as a zero added decimal number.	00, 01, …, 59
%S	Second as a zero added decimal number.	00, 01, …, 59
%p	Locale's AM or PM.	AM, PM

With this create a time stamp

- Finally print it out over screen

```
                print('{} updates - {} ({} hrs {} mins {} secs elapsed)'
                      .format(iter-1, timestamp, hrs, mins, secs))
```

- Next up

```
                #-------------------------------------------------------------
                # Validate
                #-------------------------------------------------------------

                # Validation cost
                costs = self.f_cost(*validation_data(best['other_costs']))
                z     = costs[-1] # network outputs
                costs = [float(i) for i in costs[:-1]]
```

```
                    s0    = "| validation loss / RMSE"
                    s1    = ": {:.6f} / {:.6f}".format(costs[0], costs[1])
```

remember this self.f_costs class variable we defined in __init__ execution it is theano function object.

It is being supplied with input data here

That is validation dataset, Now you will see that they have '**called**' Validation dataset class instance with parameter **'best['other_costs']' (Note:** this best_cost is empty at start **)**

This calls upon the execution of '__call__' method in dataset.py and 'Dataset' class

What call method does is, it allows class instance to behave like function and allow them to be callable

See here __call__ method

The definition of __call__ in dataset.py under Dataset class

```
    def __call__(self, best_costs, callback_results=None, update=True):
        """
        Return a batch of trials.

        best_costs : list
                     The best costs, not including the loss.

        callback_results : any, optional
                           Information used to adapt training.

        update : bool, optional
                 If `True`, return a new set of trials.

        """
        if update:
            self.update(best_costs, callback_results)
            self.ntrials += self.minibatch_size

        return [self.inputs
[:,self.trial_idx:self.trial_idx+self.minibatch_size,:],
                self.targets[:,self.trial_idx:self.trial_idx+self.minibatch_size,
:]]
```

here arguments taken are as shown, Read out comments for more clarification

by default update is true so it calls upon the update function the same class with as mentioned parameters

```
    def update(self, best_costs, callback_results):
        """
        Generate a new minibatch of trials and store them in `self.inputs` and
        `self.outputs`. For speed (but at the cost of memory), we store
`batch_size`
        trials and only create new trials if we run out of trials.

        For both inputs and outputs, the first two dimensions contain time and
        trials, respectively. For the third dimension,

          self.inputs [:,:,:Nin]  contains the inputs (including baseline and
noise),
          self.inputs [:,:,Nin:]  contains the recurrent noise,
          self.outputs[:,:,:Nout] contains the target outputs, &
          self.outputs[:,:,Nout:] contains the mask.

        Parameters
        ----------

        best_costs : list of floats
                     Performance measures, in case you want to modify the trials
                     (e.g., noise) depending on the error.

        callback_results : any
                           Results the trial function can use to modify training.

        """
```

Read comments carefully

Moving on

```
        self.trial_idx += self.minibatch_size
        if self.trial_idx + self.minibatch_size > self.batch_size:
            self.trial_idx = 0

            self.trials = []
            for b in range(self.batch_size):
                params = {
                    'callback_results': callback_results,
                    'target_output':    True,
                    'minibatch_index':  b,
                    'best_costs':       best_costs,
                    'name':             self.name
                    }
                self.trials.append(self.task.generate_trial(self.rng, self.dt,
params))
```

The self.trial_idx is a class instance variable that got its foundation in init method i.e., as soon as class object is created

If you forgot about this then please brush up from page 85-88

Next up we:

Check if trial_idx +minibatch is greater than batch_size is, if it is True:

- Then set trial_idx as zero
- Set trials list as empty (though it was already empty for now but for in case it has residues from other data)
- Run for loop varying b from 0 to batch_size-1
    - Set 'params' dict as specified in code
    - Then append a result return from generate trial function of our model we are using(S1_code for our current case), to the trials array
    - The same will repeat for the lifespan of this loop
- (It will take too many pages to show output of Trial list so it will be skipped)
-

```
            # Longest trial
            k = np.argmax([len(trial['t']) for trial in self.trials])
            t = self.trials[k]['t']
```

numpy.argmax(*a, axis=None, out=None, *, keepdims=<no value>*)[source]

Returns the indices of the maximum values along an axis.

Parameters

a : *array_like*

Input array.

axis : *int, optional*

By default, the index is into the flattened array, otherwise along the specified axis.

out : *array, optional*

If provided, the result will be inserted into this array. It should be of the appropriate shape and dtype.

keepdims: *bool, optional*

If this is set to True, the axes which are reduced are left in the result as dimensions with size one. With this option, the result will broadcast correctly against the array.

Returns

index_array : *ndarray of ints*

Array of indices into the array. It has the same shape as *a.shape* with the dimension along *axis* removed.

If *keepdims* is set to True, then the size of *axis* will be 1 with the resulting array having same shape as *a.shape*.

'k' and 't' defined as shown

k is index of the maximum length any element of trial

t is time step array at index [k] in trial array

Two Sample k and t:

```
k is :  0
t is :  [  20.   40.   60.   80.  100.  120.  140.  160.  180.  200.  220.  240.
  260.  280.  300.  320.  340.  360.  380.  400.  420.  440.  460.  480.
  500.  520.  540.  560.  580.  600.  620.  640.  660.  680.  700.  720.
  740.  760.  780.  800.  820.  840.  860.  880.  900.  920.  940.  960.
  980. 1000. 1020. 1040. 1060. 1080. 1100. 1120. 1140. 1160. 1180. 1200.]
k is :  11
t is :  [  20.   40.   60.   80.  100.  120.  140.  160.  180.  200.  220.  240.
  260.  280.  300.  320.  340.  360.  380.  400.  420.  440.  460.  480.
  500.  520.  540.  560.  580.  600.  620.  640.  660.  680.  700.  720.
  740.  760.  780.  800.  820.  840.  860.  880.  900.  920.  940.  960.
  980. 1000. 1020. 1040. 1060. 1080. 1100. 1120. 1140. 1160. 1180. 1200.
 1220. 1240. 1260. 1280. 1300. 1320. 1340. 1360. 1380. 1400. 1420. 1440.
 1460. 1480. 1500. 1520. 1540. 1560. 1580. 1600. 1620. 1640. 1660. 1680.
 1700. 1720. 1740. 1760. 1780. 1800. 1820. 1840. 1860. 1880. 1900. 1920.
 1940. 1960. 1980. 2000.]
```

```
            # Input and output matrices
            Nin, N, Nout = self.network_shape
            T = len(t)
            B = self.batch_size
            x = np.zeros((T, B, Nin+N),  dtype=self.floatX)
            y = np.zeros((T, B, 2*Nout), dtype=self.floatX)
```

- Take out network shape store it in shown above 3 variables
- T length of time step array that at index of maximum length trial
- B – batch_size
- x is 3-D zero matrix of size T x B x (Nin+N) and datatype self.floatx (It will store inputs)
- y is 3-D zero matrix of size T x B x 2**Nout and same dtype as above (It will store outputs)

```
            # Pad trials
            for b, trial in enumerate(self.trials):
                Nt = len(trial['t'])
                if Nin > 0:
                    x[:Nt,b,:Nin] = trial['inputs']
                y[:Nt,b,:Nout] = trial['outputs']
                if 'mask' in trial:
                    y[:Nt,b,Nout:] = trial['mask']
                else:
                    y[:Nt,b,Nout:] = 1
```

- enumerate provides every element of self.trials with a counter that starts from 0 i.e, the way we are using it we will get b from 0 to len(trial)-1 with every element trial from self.trials
- This will run a for loop for every array element in self.trials list
    - Nt will be length of current trial element
    - If Nin is greater than zero
        - From index 0 to Nt along axis 0, at index b along axis 1 and from index 0 to Nin, in numpy array x put the input array of the trial element
        - Note: If you visualise in mind, In 3-D space we fix y and take a plane X-Z, i.e 2-D space of size Nt by Nin and Input array which is also of size Nt by Nin and change values in X-Z plane to repective values from inputs array
    - Change the X-Z plane of y to accommodate outputs similar to done for inputs
    - If mask is to be used
        - Change the X-Z plane but this time Take z-axis from Nout to end and set there the mask from trial
    - If mask not to be used
        - Change the above mentioned plan to have all 1's
    - End of the loop
- Moving on

```
            # Input noise
            if Nin > 0:
                if np.isscalar(self.var_in) or self.var_in.ndim == 1:
                    # Independent noise
                    if np.any(self.var_in > 0):
                        r = np.sqrt(self.var_in)*self.rng.normal(size=(T, B,
Nin))
                    else:
                        r = 0
                else:
                    # Correlated noise
                    r = self.rng.multivariate_normal(np.zeros(Nin), self.var_in,
(T, B))
                x[:,:,:Nin] += self.baseline_in + r
```

- If Nin is greater than zero
    - Check if variance in input is scalar or atleast has dimension 1
        - Check whether there is any var_in greater than 0 along any axis
            - Set 'r' as square root of var_in multiplied normal distribution of size T, B, Nin
            - This multiplication will be vector multiplication
        - Else
            - Set r as zero

- Else
    - Set r as multivariate normal distribution drawn with mean as np.zeros(Nin), covariance as self.var_in and size as T x B
- Add baseline input plus r with third axis of x from 0 to Nin
- End of If condition of Nin

- Moving on

```
            # Recurrent noise
            if np.isscalar(self.var_rec) or self.var_rec.ndim == 1:
                # Independent noise
                if np.any(self.var_rec > 0):
                    r = np.sqrt(self.var_rec)*self.rng.normal(size=(T, B, N))
                else:
                    r = 0
            else:
                # Correlated noise
                r = self.rng.multivariate_normal(np.zeros(N), self.var_rec, (T,
B))
            x[:,:,Nin:] = np.sqrt(self.tau)*r
```

- Similar as Input Noise

```
            # Keep inputs positive
            if self.rectify_inputs:
                x[:,:,:Nin] = Dataset.rectify(x[:,:,:Nin])

            # Save
            self.inputs  = x
            self.targets = y
```

- Units are to be rectified
    - Call Dataset.rectify it will return every value that are positive as it is while negative ones will be set as zero
    - Definition of rectify

```
    @staticmethod
    def rectify(x):
        return x*(x > 0)
```

    - It used conditional argument (x>0) is it is true then consider it as 1 else 0
- Finally save x and y as class variables inputs and targets
- End of Update function return back to __call__ execution

```
        if update:
            self.update(best_costs, callback_results)
            self.ntrials += self.minibatch_size

        return[self.inputs[:,self.trial_idx:self.trial_idx+self.minibatch_size,:]
,       self.targets[:,self.trial_idx:self.trial_idx+self.minibatch_size,:]]
```

Now they increase count of ntrials class variable by size of minibatch

Finally we return an array with first element as slice of self.inputs with whole axis 0, axis 1 from trial_idx to trial_idx +minibatch_size and whole axis 2

And second element as self.targets sliced in similar fashion as inputs

End of __call__ execution

Return back to SGD.py

```
                    #-------------------------------------------------------------
                    # Validate
                    #-------------------------------------------------------------

                    # Validation cost
                    costs = self.f_cost(*validation_data(best['other_costs']))
```

The received Validation Data is operated by f_cost function That we defined in init execution of dataset.py

```
        # Cost function
        self.f_cost = theanotools.function(inputs, [costs[0] + regs] + costs[1:]
+ [z])
```

So, using the inputs we gave as validation dataset it will calculate the second argument function

Now as you have seen our return from __call__ was an array of two elements and if you recall inputs was passed on from trainer.py where it was defined as

```
        # Input-output pairs
        inputs = [u, target]
```

and u, target were defined as

```
        # (time, trials, outputs)
        target = T.tensor3('target')
```

(both are present in trainer.py between line 600-700)

```
        u    = T.tensor3('u')
```

so the data provided by us will give value to these tensors

and if you look at the definition of outputs elements like costs, regs and z(look for them in trainer.py) you will see their clear relation with target and u which ultimately will be used to calculate them and finally they will be process through as specified in output argument and returned and stored as 'costs'

A sample Validation data is given at the end of this documentation

Moving on

```
                    z     = costs[-1] # network outputs
                    costs = [float(i) for i in costs[:-1]]
                    s0    = "| validation loss / RMSE"
                    s1    = ": {:.6f} / {:.6f}".format(costs[0], costs[1])
```

costs we get is a list

```
Costs prev :  [array(0.08331212, dtype=float32), array(0.28863838,
dtype=float32), array([[[0.16242363, 0.17173481],
        [0.21645102, 0.23772119],
        [0.23057163, 0.20331168],
        ...,
        [0.2089929 , 0.18480404],
        [0.18901555, 0.2072623 ],
        [0.17519335, 0.21910541]],

       [[0.14191571, 0.13151848],
        [0.2469007 , 0.2679421 ],
        [0.26239184, 0.23697478],
        ...,
        [0.22562318, 0.1945677 ],
        [0.18398911, 0.23181675],
        [0.22291248, 0.2830194 ]],

       [[0.16159974, 0.12494648],
        [0.22041921, 0.21349387],
        [0.21285857, 0.21738899],
        ...,
        [0.25757986, 0.22684097],
        [0.22013164, 0.29186028],
        [0.19627085, 0.26166078]],

       ...,

       [[0.4147702 , 0.48098513],
        [0.31522083, 0.44148117],
        [0.24822114, 0.38742855],
        ...,
        [0.30423552, 0.42251644],
```

```
        [0.2830248 , 0.4348571 ],
        [0.35698384, 0.55772746]],

       [[0.41537333, 0.4721934 ],
        [0.3339822 , 0.4267443 ],
        [0.208784  , 0.37522888],
        ...,
        [0.3145218 , 0.4554906 ],
        [0.28816697, 0.44989747],
        [0.365402  , 0.5326369 ]],

       [[0.39303258, 0.49883103],
        [0.32020915, 0.3916697 ],
        [0.20259742, 0.34679753],
        ...,
        [0.329787  , 0.45131227],
        [0.27056548, 0.44637948],
        [0.36522907, 0.5628662 ]]], dtype=float32)]
```

so costs[-1] will refer to last of the elements i.e., second

then overwrite costs with a list of float(i)'s for every in costs[:-1]

the cost becomes

```
Costs :  [0.08331212401390076, 0.2886383831501007]
```

costs[:-1] refers to elements of costs starting from index 0 to last index -1 i.e., everything except for last element

s0 is string as shown

s1 is also a string with cost[0] and cost[1] kept in string format til 6 decimal places

```
s1 :  : 0.083312 / 0.288638
```

```
                    # Dashes
                    nfill = 70
```

Will be used in decoration

```
                # Compute task-specific performance
                if performance is not None:
                    costs.append(performance(validation_data.get_trials(),
                                             SGD.get_value(z)))
                    s0    += " / performance"
                    s1    += " / {:.2f}".format(costs[-1])
                s = s0 + s1
```

same as comments says

if performance if not None

- Append to costs list the return from performance function (variable defined in S1_code and has been since carried over by params dictionary)

In task tools performance_2afc definition

```
def performance_2afc(trials, z):
    ends    = [len(trial['t'])-1 for trial in trials]
    choices = [np.argmax(z[ends[i],i]) for i, end in enumerate(ends)]
    correct = [choice == trial['info']['choice']
               for choice, trial in zip(choices, trials) if trial['info']]

    return 100*sum(correct)/len(correct)
```

  - The parameters supplied to performance function are validation_data.get_trials() and SGD.get_values(z) from function pov these are trials and z
  - Validation_data.get_trials definition

```
    def get_trials(self):
        """
        Return info for current trials.

        """
        return self.trials[self.trial_idx:self.trial_idx+self.minibatch_size]
```

returns self.trial slice as shown (self.trial defined in update call from__call__ method when we generated ‘costs’ with theano function f_costs)

  - z is outputs that was taken from last element of ‘costs’ that was generated from f_costs it goes through get_value function of same SGD.py
  - definition of get value

```
    @staticmethod
    def get_value(x):
        """
        Return the numerical value of `x`.

        """
        if hasattr(x, 'get_value'):
            return x.get_value(borrow=True)
        if hasattr(x, 'eval'):
            return x.eval()
        return x
```

returns the value of z in numerical format ()

- Hence the costs list will be updated with performance
- Then s1 and s0 are strings as shown, s1 contains last element of costs as string i.e., latest performance

Moving on

```
                    # Callback
                    if self.p['callback'] is not None:
                        callback_results = self.p['callback'](
                            validation_data.get_trials(), SGD.get_value(z)
                            )
                    else:
                        callback_results = None
```

If validation dataset has to be evaluated then callback will be used

So if yes then callback function will be applied similarly to performance function

Otherwise set callback_results to None

```
                    # Keep track of costs
                    costs_history.append((gradient_data.ntrials, costs))
```

Cost_history getting updated

```
                    # Record the value of the regularization term in the last
iteration
                    if tr_Omega is not None:
                        Omega_history.append(
                            (gradient_data.ntrials, lambda_Omega*tr_Omega)
                            )
```

Omega history getting updated, if tr_Omega is not none that is

```
                    # New best
                    if costs[0] < best['cost']:
                        s += ' ' + '-'*(nfill - len(s))
                        s += " NEW BEST (prev. best:
{:.6f})".format(best['cost'])
                        best = {
                            'iter':        iter,
                            'cost':        costs[0],
                            'other_costs': costs[1:],
                            'params':      SGD.get_values(self.save_values)
                            }
                    print(s)
```

if current cost's zeroth element is less than best cost than print out in decorative format that New best is found

and update 'best' dictionary

Nothing much to explain here

Note: *reminder we are inside a for loop*

```
                    # Spectral radius
                    rho = RNN.spectral_radius(self.Wrec_.eval())
```

using recurrent weights evaluate rho, the spectral radius

Definition of Spectral_radius function

```
    @staticmethod
    def spectral_radius(A):
        """
        Compute the spectral radius of a matrix.

        """
        return np.max(abs(np.linalg.eigvals(A)))
```

np.max() evaluates max of an ndarray (N-dimensional array)

abs() calculates absolute numerical value of any numeral

np.linalig.eigvals() as name suggests it evalutes eigen values for a matrix. For mopre info check [here]

```
                    # Format
                    Omega = ('n/a' if tr_Omega is None
                             else '{:.8f}'.format(float(tr_Omega)))
                    gnorm = ('n/a' if tr_gnorm is None
                             else '{:.8f}'.format(float(tr_gnorm)))
```

set omega and gradient norm **string** according to if they are None or not, {:.8f}: till 8 decimal places

```
                    # Info
                    print("| Omega      (last iter) = {}".format(Omega))
                    print("| grad. norm (last iter) = {}".format(gnorm))
                    print("| rho                    = {:.8f}".format(rho))
                    sys.stdout.flush()
```

sample output:

```
|validation loss / RMSE / performance: 0.083312 / 0.288638 / 57.41 --- NEW BEST (prev. best: 0.089410)
| Omega      (last iter) = 0.00885053
| grad. norm (last iter) = 0.50862914
| rho                    = 2.25880814
```

```
#-------------------------------------------------------------
# Save progress
#-------------------------------------------------------------

save = {
    'params':         {k: v for k, v in self.p.items()
                       if k not in save_exclude},
    'varlist':        self.trainable_names,
    'iter':           iter,
    'current':        SGD.get_values(self.trainables),
    'best':           best,
    'costs_history':  costs_history,
    'Omega_history':  Omega_history,
    'rng_gradient':   gradient_data.rng,
    'rng_validation': validation_data.rng
    }
dump(savefile, save)
```

as comments describe it saves the progress

- Params : dictionary with k key and v value for k and b in self.p.items() with a condition imposed that k should not be in save_exclude
- Varlist: array of trainable names
- Iter : the looping element that we are increasing every iteration of loop
- All else are self-explanatory and already encountered variables left for reader to see for themselves

dump() stores the save dictionary with name savefile on local storage

next up is conditons to break from the loop and fin final the training step for the iteration

```
if costs[1] <= self.p['min_error']:
    print("Reached minimum error of {:.6f}"
          .format(self.p['min_error']))
    break
```

if current error less than minimum error print as specified and break the loop

```
                    # This termination criterion assumes that performance is not
None
                    if terminate(np.array([c[-1] for _, c in costs_history])):
                        print("Termination criterion satisfied -- we\'ll call it
a day.")
                        break
```

use terminate function defined the model file carried over from params to evaluate if termination criterion is satisfied if it is then break loop

terminate function in S1_code.py

```
def terminate(performance_history):
    return np.mean(performance_history[-5:]) > 85
```

the element supplied to terminate sunction is numpy array of let's some variable temp to remaining stuff

so temp is [c[-1] for _,c in costs_history], temp is a list

a sample costs history looks like this:

```
Cost_history= [(0, [0.18540038168430328, 0.4305814504623413,
49.848637739656915]), (0, [0.2004518061876297, 0.44771844148635864, 53.01085883514314]), (0,
[0.18339714407920837, 0.42824894189834595, 51.4543630892678]), (10000, [0.10977434366941452,
0.3313221037387848, 49.749749749749746]), (20000, [0.09377618134021759, 0.30622896552085876,
48.60834990059642]), (0, [0.08975014835596085, 0.2995832860469818, 49.447236180904525]), (10000,
[0.0914217159152031, 0.3023602366447449, 53.90702274975272]), (20000, [0.08941011875867844,
0.29901525378227234, 53.90702274975272]), (30000, [0.08331212401390076, 0.2886383831501007,
57.414829659318634]), (0, [0.07830247282981873, 0.27982577681541443, 56.80655066530194]), (10000,
[0.07940338551998138, 0.2817860543727875, 64.6]), (20000, [0.06881888955831528, 0.26233354210853577,
71.9438877755511]), (30000, [0.059132691472768784, 0.24317213892936707, 75.57788944723617]), (40000,
[0.058723486959934235, 0.24232929944992065, 77.0392749244713])]
```

And a corresponding temp looks like

```
Temp = [49.848637739656915, 53.01085883514314, 51.4543630892678,
49.749749749749746, 48.60834990059642, 49.447236180904525, 53.90702274975272,
53.90702274975272, 57.414829659318634, 56.80655066530194, 64.6, 71.9438877755511,
75.57788944723617, 77.0392749244713]
```

So the termination function checks for last 5 values average if > 85 as termination criteria

If satisfied Training closed

```
            if iter - best['iter'] > patience:
                print("We've run out of patience -- time to give up.")
                break
```

if iter subtracted from best iteration is greater than patience then it also terminates the training

```
            #-----------------------------------------------------------------
            # Training step
            #-----------------------------------------------------------------

            tr_cost, tr_gnorm, tr_Omega, tr_nelems, tr_x = self.train_step(
                *(gradient_data(best['other_costs'], callback_results)
                  + [alpha, lambda_Omega, lr, maxnorm, bound])
                 )
```

Lastly the Most important step for the whole loop cycle the updating of training cost, omega, gnorm , x and nelems which is done through the train_step function we defined in init og SGD.py

The logic is still same as f_costs

The provided inputs will be used to calculate the specified outputs

So lets check definition of train_step

```
        # Update function
        self.train_step = theanotools.function(
            inputs + [alpha, lambda_Omega, lr, maxnorm, bound],
            [costs[0] + regs, gnorm, Omega, nelems, x],
            updates=updates
            )
```

On inspection:

- inputs is (gradient_data(best['other_costs'], callback_results)
- the array after '+' is equivalent to similar name in definition
- the output we calculate is array with 5 elements
    - the first 'cost[0]+regs' will be stored as 'tr_costs' in output
    - rest all have respective similar names and will be stored respectively in order

Now, gradient_data(best['other_costs'], callback_results) will have same execution we discussed for validation data so please refer to page 121-128

The result will be a similar dataset similar validataion dataset attached to end of documentation along with sample save file

Defition of variables follow through them and you will see inner calculctions with the specified input

Present in trainer.py

```
        u    = T.tensor3('u')
```

```
        target = T.tensor3('target')
```

```
        # Input-output pairs
        inputs = [u, target]
```

```
        costs = [loss, error]
```

```
        # Loss, not including the regularization terms
        loss = T.sum(f_loss(z, target[:,:,:Nout])*mask)/masknorm

        # Root-mean-squared error
        error = T.sqrt(T.sum(theanotools.L2(z, target[:,:,:Nout])*mask)/masknorm)
```

present in sgd.py

```
        g = []
        if 'Win' in self.trainable_names:
            g += [g_Win]
        g += [g_Wrec, g_Wout]
        if 'brec' in self.trainable_names:
            g += [g_brec]
        if 'bout' in self.trainable_names:
            g += [g_bout]
        if 'x0' in self.trainable_names:
            g += [g_x0]

        # Clip
        gnorm = T.sqrt(sum([(i**2).sum() for i in g]))
```

```
lambda_Omega = T.scalar('lambda_Omega')
lr           = T.scalar('lr')
maxnorm      = T.scalar('maxnorm')
bound        = T.scalar('bound')
```

```
Omega = (T.switch(T.ge(denom, bound), num/denom, 1) - 1)**2
```

```
nelems = T.mean(T.ge(denom, bound), axis=1)
```

Follow through the code yourself to better grasp the definitions

See theano.function : ignore compile.function it is full name of theano.function

theano.compile.function.function(*inputs, outputs, mode=None, updates=None, givens=None, no_default_updates=False, accept_inplace=False, name=None, rebuild_strict=True, allow_input_downcast=None, profile=None, on_unused_input='raise'*)[source]

Return a `callable object` that will calculate *outputs* from *inputs*.

Parameters:

- **params** (*list of either Variable or In instances, but not shared variables.*) – the returned `Function` instance will have parameters for these variables.
- **outputs** (*list of Variables or Out instances*) – expressions to compute.
- **mode** (None, string or `Mode` instance.) – compilation mode
- **updates** (*iterable over pairs (shared_variable, new_expression) List, tuple or dict.*) – expressions for new `SharedVariable` values
- **givens** (*iterable over pairs (Var1, Var2) of Variables. List, tuple or dict. The Var1 and Var2 in each pair must have the same Type.*) – specific substitutions to make in the computation graph (Var2 replaces Var1).
- **no_default_updates** (*either bool or list of Variables*) – if True, do not perform any automatic update on Variables. If

False (default), perform them all. Else, perform automatic updates on all Variables that are neither in `updates` nor in `no_default_updates`.

- **name** – an optional name for this function. The profile mode will print the time spent in this function.
- **rebuild_strict** – True (Default) is the safer and better tested setting, in which case *givens* must substitute new variables with the same Type as the variables they replace. False is a you-better-know-what-you-are-doing setting, that permits *givens* to replace variables with new variables of any Type. The consequence of changing a Type is that all results depending on that variable may have a different Type too (the graph is rebuilt from inputs to outputs). If one of the new types does not make sense for one of the Ops in the graph, an Exception will be raised.
- **allow_input_downcast** (*Boolean or None*) – True means that the values passed as inputs when calling the function can be silently downcasted to fit the dtype of the corresponding Variable, which may lose precision. False means that it will only be cast to a more general, or precise, type. None (default) is almost like False, but allows downcasting of Python float scalars to floatX.
- **profile** (*None, True, or ProfileStats instance*) – accumulate profiling information into a given ProfileStats instance. If argument is *True* then a new ProfileStats instance will be used. This profiling object will be available via self.profile.
- **on_unused_input** – What to do if a variable in the 'inputs' list is not used in the graph. Possible values are 'raise', 'warn', and 'ignore'.

Return type: `Function` instance

Returns: a callable object that will compute the outputs (given the inputs) and update the implicit function arguments according to the *updates*.

So at this all whole process will repeat for increasing ‘iter’ and the training will continue until meets any termination criterion

Lastly after the training will we will go back to trainer.py where the sgd.train was initiated

We find we were at the end of trainer.py

So, we go back to model.py there we also are now at end

So, we go back to S1_code.py thus ending the execution of training

```
    # Run the trained network with 16*3.2% = 51.2% coherence for choice 1
    rnn        = RNN('savefile.pkl', {'dt': 0.5})
    trial_func = generate_trial
    trial_args = {'name': 'test', 'catch': False, 'coh': 16, 'left_right': 1}
    info       = rnn.run(inputs=(trial_func, trial_args))
    # rnn.plot
    print(info)
```

Final lines that if you follow through the whole guide faithfully can figure out on your own

Just an explanation of some new terms you will encounter

```
OrderedDict()
```
See here

Basically a simple dictionary but one that will remember order of insertion

```
.astype()
```
See here

```
 np.concatenate()
```
See here

What we in the end get from these lines is:

```
[ pycog.rnn.RNN ] 10500 updates, best error = 0.18983847, spectral radius =
2.37324977
=> settings
   | dt:        0.5 ms
   | threshold: 0.0001
```

And the info at last prints out as :

```
{'coh': 16, 'left_right': 1, 'choice': 0, 'epochs': {'fixation': (0, 100),
'stimulus': (100, 900), 'decision': (900, 1200), 'T': 1200}}
```

_____THE END OF THE DOCUMENTATION _____

ENDING NOTE:

There's another way to run this training regime by using do.py

Just go to terminal do initial setup then move over to directory with do.py and type 'python do.py S1_code.py train'

Sample Validation dataset

```
validatio_data:  [array([[[ 1.87356815e-01,  2.24577859e-01, -6.82947516e-01, ...,
         -1.16603673e+00, -2.76309699e-01,  3.36594999e-01],
        [ 2.02078342e-01,  1.96003541e-01,  4.39434677e-01, ...,
         -1.74473837e-01, -1.09302700e+00,  8.18096921e-02],
        [ 1.96562067e-01,  1.62422374e-01, -3.60777564e-02, ...,
         -2.78087318e-01,  2.16506615e-01, -2.13688076e-01],
        ...,
        [ 2.45321482e-01,  1.86359346e-01,  2.83377945e-01, ...,
         -2.39409804e-01,  2.44604751e-01, -7.52366111e-02],
        [ 2.67792523e-01,  2.30574325e-01,  6.10327780e-01, ...,
          3.06165546e-01, -3.68651539e-01, -1.38195992e-01],
        [ 1.59658358e-01,  1.71731263e-01,  8.88626218e-01, ...,
          3.91458452e-01,  1.61627084e-01,  2.93352991e-01]],

       [[ 1.81575179e-01,  2.06386566e-01,  5.73951602e-01, ...,
         -1.85103834e-01,  5.67481935e-01, -8.32273960e-02],
        [ 2.14691058e-01,  2.19377890e-01,  6.08715520e-04, ...,
          5.77196658e-01,  5.66614449e-01,  1.20743047e-02],
        [ 1.95194036e-01,  2.15208769e-01, -1.35138363e-01, ...,
         -2.41146028e-01,  1.28201795e+00,  6.47300363e-01],
        ...,
        [ 2.44644284e-01,  2.12605059e-01, -1.83408588e-01, ...,
          7.82462895e-01, -2.61443853e-01,  7.72697508e-01],
        [ 2.66932219e-01,  2.15308264e-01,  2.80773461e-01, ...,
         -2.01321319e-01,  3.59785110e-01, -1.25405520e-01],
        [ 2.40223408e-01,  1.46342427e-01, -1.34875759e-01, ...,
          2.99343020e-01,  7.98132658e-01, -1.35660291e-01]],

       [[ 1.95054591e-01,  1.84104919e-01,  1.09025083e-01, ...,
         -2.00914904e-01,  3.17184269e-01, -2.19535440e-01],
        [ 1.92269921e-01,  1.62766203e-01, -1.48440212e-01, ...,
         -7.68754482e-02, -2.19636038e-01,  1.20928511e-01],
        [ 2.19286546e-01,  1.52213424e-01, -5.29166698e-01, ...,
          1.23203740e-01,  3.85545909e-01, -2.17437238e-01],
```

```
       ...,
       [ 2.59840518e-01,  2.52489835e-01, -6.37640476e-01, ...,
         8.30623806e-01,  6.30663276e-01, -3.86543274e-01],
       [ 2.32459590e-01,  1.80840373e-01,  5.86713910e-01, ...,
        -2.45951384e-01, -7.88654685e-01, -1.54583365e-01],
       [ 2.51993090e-01,  1.98983639e-01, -3.43625784e-01, ...,
         3.11093181e-01, -5.44647276e-02, -4.46719863e-02]],

      ...,

      [[ 1.99203581e-01,  2.36112550e-01, -5.32302320e-01, ...,
        -3.55627149e-01, -2.95001119e-01, -4.15558964e-01],
       [ 1.78926736e-01,  2.13591650e-01,  5.08006334e-01, ...,
        -3.29539269e-01,  2.69288182e-01, -6.84668779e-01],
       [ 1.41056225e-01,  2.01147124e-01,  4.16566543e-02, ...,
        -5.80951691e-01,  1.16309486e-01, -2.64913172e-01],
       ...,
       [ 2.08144456e-01,  2.55931556e-01,  3.64044100e-01, ...,
         5.55689037e-01, -2.61455148e-01, -6.10113323e-01],
       [ 1.58497289e-01,  2.11359620e-01,  3.08432043e-01, ...,
        -3.32134124e-03,  8.46854329e-01, -4.13965762e-01],
       [ 1.92761987e-01,  2.07479566e-01, -2.60469794e-01, ...,
        -6.58154666e-01,  5.52036941e-01,  5.63779712e-01]],

      [[ 2.36226231e-01,  1.96742803e-01,  1.18313506e-01, ...,
         3.59808832e-01, -2.33020738e-01,  5.13048351e-01],
       [ 1.48622021e-01,  2.02167377e-01,  5.35113811e-01, ...,
         4.07603145e-01,  1.24071980e+00,  1.82832982e-02],
       [ 1.92579091e-01,  1.88075557e-01, -6.11658216e-01, ...,
        -9.13224518e-02, -1.02388275e+00, -2.47111529e-01],
       ...,
       [ 1.84728086e-01,  1.89027607e-01,  1.55823767e-01, ...,
        -2.79348522e-01, -1.59064651e-01, -7.13235199e-01],
       [ 1.42427504e-01,  1.92292541e-01,  9.14897397e-03, ...,
        -4.35662307e-02,  5.44772267e-01, -1.27402782e-01],
       [ 2.13727087e-01,  1.89516678e-01,  4.83819157e-01, ...,
        -6.95433199e-01, -1.07295215e+00, -1.76365584e-01]],

      [[ 1.85653165e-01,  1.30373284e-01,  4.17890161e-01, ...,
        -2.34031573e-01, -2.85237908e-01,  8.99446905e-02],
       [ 1.12634912e-01,  1.36636287e-01, -6.27103627e-01, ...,
        -6.40734196e-01, -1.44459248e-01, -3.08375955e-01],
       [ 1.94034487e-01,  1.63235307e-01,  2.44584620e-01, ...,
        -3.04846078e-01,  4.18361962e-01, -1.21190488e-01],
       ...,
```

```
       [ 2.06802353e-01,  2.12803736e-01,  3.01256061e-01, ...,
        -2.57740408e-01,  4.31607991e-01,  4.64798152e-01],
       [ 1.76212013e-01,  2.08417818e-01,  1.84098631e-01, ...,
        -3.84957165e-01, -6.01008296e-01,  1.68779284e-01],
       [ 1.89909160e-01,  2.07519844e-01,  2.57156670e-01, ...,
        -5.36775440e-02, -3.60311568e-02,  5.73692262e-01]]],
     dtype=float32), array([[[0.2, 0.2, 1. , 1. ],
       [0.2, 0.2, 1. , 1. ],
       [0.2, 0.2, 1. , 1. ],
       ...,
       [0.2, 0.2, 1. , 1. ],
       [0.2, 0.2, 1. , 1. ],
       [0.2, 0.2, 1. , 1. ]],

      [[0.2, 0.2, 1. , 1. ],
       [0.2, 0.2, 1. , 1. ],
       [0.2, 0.2, 1. , 1. ],
       ...,
       [0.2, 0.2, 1. , 1. ],
       [0.2, 0.2, 1. , 1. ],
       [0.2, 0.2, 1. , 1. ]],

      [[0.2, 0.2, 1. , 1. ],
       [0.2, 0.2, 1. , 1. ],
       [0.2, 0.2, 1. , 1. ],
       ...,
       [0.2, 0.2, 1. , 1. ],
       [0.2, 0.2, 1. , 1. ],
       [0.2, 0.2, 1. , 1. ]],
      ...,

      [[0. , 0. , 0. , 0. ],
       [0. , 0. , 0. , 0. ],
       [0. , 0. , 0. , 0. ],
       ...,
       [0. , 0. , 0. , 0. ],
       [0. , 0. , 0. , 0. ],
       [0. , 0. , 0. , 0. ]],

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

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

      [[0. , 0. , 0. , 0. ],
       [0. , 0. , 0. , 0. ],
       [0. , 0. , 0. , 0. ],
       ...,
       [0. , 0. , 0. , 0. ],
       [0. , 0. , 0. , 0. ],
       [0. , 0. , 0. , 0. ]]], dtype=float32)
```

Sample Save file output

```
save is:  {'params': {'Nin': 2, 'N': 100, 'Nout': 2, 'Cout': <pycog.connectivity.Connectivity
object at 0x00000247C0E8FE20>, 'ei': array([
1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,
       1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,
       1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,
       1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,
       1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1, -1, -1, -1, -1, -1,
      -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1]), 'n_validation': 1100,
'extra_info': {}, 'rectify_inputs': True, 'train_brec': False, 'brec': 0, 'train_bout': False,
'bout': 0, 'train_x0': True, 'x0': 0.1, 'mode': 'batch', 'tau': 100, 'tau_in': 100, 'Cin':
None, 'Crec': <pycog.connectivity.Connectivity object at 0x00000247C0E8FD30>,
'ei_positive_func': 'rectify', 'hidden_activation': 'rectify', 'output_activation': 'linear',
'n_gradient': 20, 'gradient_batch_size': None, 'validation_batch_size': None, 'lambda_Omega':
2, 'lambda1_in': 0, 'lambda1_rec': 0, 'lambda1_out': 0, 'lambda2_in': 0, 'lambda2_rec': 0,
'lambda2_out': 0, 'lambda2_r': 0, 'min_error': 0, 'learning_rate': 0.01, 'max_gradient_norm':
1, 'bound': 1e-20, 'baseline_in': 0.2, 'var_in': 0.0001, 'var_rec': 0.0225, 'seed': 1234,
'gradient_seed': 11, 'validation_seed': 22, 'structure': {}, 'rho0': 1.5, 'max_iter':
10000000, 'dt': 20.0, 'distribution_in': 'uniform', 'distribution_rec': 'gamma',
'distribution_out': 'uniform', 'gamma_k': 2, 'checkfreq': None, 'patience': None, 'momentum':
False, 'method': 'sgd'}, 'varlist': ['Win', 'Wrec', 'Wout', 'x0'], 'iter': 10001, 'current':
[array([[ 1.11483477e-01,  1.92378648e-02],
       [ 3.01854927e-02,  8.42370540e-02],
       [ 5.98337688e-02,  7.50019699e-02],
       [-7.12634937e-06,  2.25062400e-01],
       [ 1.33209497e-01,  9.81266499e-02],
       [ 4.08531055e-02,  8.80653411e-03],
       [ 3.29886898e-02,  6.89789951e-02],
       [ 3.38914879e-02,  2.15001404e-01],
       [-1.85713452e-06,  3.51810694e-01],
       [-9.73172064e-05,  3.28007370e-01],
       [ 2.38535404e-02,  4.79265675e-02],
       [ 9.44308564e-02,  1.53051736e-02],
       [ 6.40549511e-02,  1.44989312e-01],
```

```
      [ 3.93945053e-02,  7.07589015e-02],
      [-6.42209197e-05,  1.73065767e-01],
      [ 6.72387853e-02,  3.02629806e-02],
      [ 7.50776380e-02,  8.97782892e-02],
      [ 3.45710218e-02,  6.67635873e-02],
      [ 7.86113814e-02,  1.18180431e-01],
      [ 9.93348937e-03,  1.52366489e-01],
      [ 2.78285980e-01,  3.58926244e-02],
      [ 8.88623148e-02,  4.33318019e-02],
      [ 3.55960727e-02,  3.96667235e-03],
      [ 1.00297630e-01,  1.26457717e-02],
      [ 4.92543739e-04,  1.23981729e-01],
      [ 8.09945837e-02,  3.98963504e-02],
      [ 2.23153215e-02,  9.53240395e-02],
      [-1.75353503e-04,  3.77375484e-01],
      [ 5.88899739e-02,  1.27746999e-01],
      [ 1.12518996e-01,  7.48849437e-02],
      [ 4.38743159e-02,  4.16703895e-02],
      [ 6.38468564e-03,  7.11650699e-02],
      [ 7.59565830e-02,  2.27779061e-01],
      [-1.44295685e-04,  3.68534178e-01],
      [ 1.70470491e-01,  2.34196693e-01],
      [ 5.83213940e-02,  3.05309352e-02],
      [ 7.20524509e-03,  1.49210811e-01],
      [ 1.57376766e-01,  2.07185894e-01],
      [ 1.64468847e-02,  1.03404699e-02],
      [ 3.91270936e-01, -1.13925838e-04],
      [ 2.47511148e-01, -7.16336071e-05],
      [ 9.43506211e-02,  9.88454148e-02],
      [ 1.54151088e-02,  1.09389983e-01],
      [ 9.04835314e-02,  1.03662580e-01],
      [ 2.71605421e-02,  2.64651384e-02],
      [ 9.23164636e-02,  3.42725776e-02],
      [-1.88821141e-05,  2.08290666e-01],
      [-1.94249515e-06,  2.17942446e-01],
      [-7.20303506e-05,  5.18092453e-01],
      [ 1.84405476e-01, -1.82334988e-05],
      [ 7.76647180e-02,  2.13509634e-01],
      [ 6.19360395e-02,  1.75038520e-02],
      [-7.15733768e-05,  3.37542921e-01],
      [ 7.48263747e-02,  1.66284330e-02],
      [ 7.01982602e-02,  1.00644037e-01],
      [ 5.34543060e-02,  4.69659679e-02],
      [-2.48408469e-07,  3.75528693e-01],
      [ 3.21293324e-01,  2.53656805e-01],
```

```
       [ 1.83410764e-01, -1.03294369e-05],
       [ 2.76860118e-01,  7.41430670e-02],
       [ 1.17539823e-01,  1.93728000e-01],
       [ 8.01449791e-02,  3.18565071e-02],
       [-3.21477273e-05,  2.91028738e-01],
       [-2.88278643e-05,  1.05662398e-01],
       [ 2.16982961e-01,  2.02908725e-01],
       [ 3.48375030e-02,  4.90755923e-02],
       [ 2.44770110e-01,  9.53757018e-02],
       [ 5.48444502e-02,  7.48098716e-02],
       [ 1.80821478e-01,  2.35227630e-01],
       [ 3.50163318e-02,  7.94453174e-02],
       [ 2.27736488e-01,  1.20620303e-01],
       [ 9.51260105e-02,  2.85057705e-02],
       [ 6.41506091e-02,  8.07185695e-02],
       [ 2.70813912e-01,  7.61970207e-02],
       [ 1.95579343e-02,  5.53474016e-02],
       [ 2.83693671e-01,  2.18431260e-02],
       [ 5.32007180e-02,  8.66350308e-02],
       [ 8.18372741e-02,  3.83403659e-01],
       [ 1.48443729e-01,  1.22247547e-01],
       [ 1.46493465e-01, -9.10309609e-07],
       [-3.85623425e-05, -2.89486343e-05],
       [ 4.41473663e-01, -1.07516156e-04],
       [ 5.64147711e-01, -2.16730186e-05],
       [-8.32871592e-06, -1.22861704e-04],
       [-8.00623093e-05, -4.51079250e-05],
       [ 5.14371216e-01, -2.34421095e-05],
       [ 6.53186362e-05,  3.49049903e-02],
       [-2.08325146e-05, -4.93327498e-05],
       [-7.36854599e-06, -6.03798753e-07],
       [-1.24100115e-04, -2.77278450e-05],
       [ 2.66690943e-02, -3.61446437e-05],
       [ 6.87408924e-01, -4.40212170e-06],
       [-7.99275585e-05,  4.80965525e-02],
       [-1.20567202e-04, -2.24581017e-05],
       [-1.14868046e-04, -1.80738934e-04],
       [-4.65498975e-04,  3.38603497e-01],
       [-3.00201296e-04,  5.00938714e-01],
       [-3.48190952e-06, -5.33804559e-05],
       [-4.01264930e-04, -3.32564159e-05],
       [-3.62234505e-06,  1.31807569e-02]], dtype=float32), array([[0.        , 0.08885651,
0.06356396, ..., 0.25410312, 0.78804827,
        0.26348507],
       [0.06306896, 0.        , 0.06821403, ..., 0.119032  , 0.25014746,
```

```
        0.39211074],
       [0.10525702, 0.02990693, 0.        , ..., 0.30997866, 0.49319315,
        0.12219751],
       ...,
       [0.02801133, 0.08948383, 0.1000785 , ..., 0.        , 0.3900255 ,
        0.27609485],
       [0.22320949, 0.08815036, 0.05537748, ..., 0.52023107, 0.        ,
        0.5003682 ],
       [0.12282094, 0.04667949, 0.04867255, ..., 0.5726365 , 0.1055176 ,
        0.        ]], dtype=float32), array([[ 2.11959276e-02,  4.00204845e-02,  3.13009582e-
02,
        -3.14415520e-05,  1.37176111e-01, -1.50983615e-05,
         1.63193829e-02, -7.08659209e-05, -8.05783129e-05,
        -2.63169459e-05,  2.09435765e-02,  1.79692190e-02,
        -9.14557313e-05,  3.55077423e-02, -1.40915217e-04,
         2.55094506e-02,  4.99977432e-02,  3.02650053e-02,
         3.41937914e-02,  6.33768812e-02,  7.00415820e-02,
         2.38701981e-02,  6.91067353e-02, -3.99341297e-05,
        -1.13500806e-04,  8.61845687e-02,  8.63438770e-02,
        -1.72774045e-04,  6.25679046e-02,  1.23451920e-02,
         2.93068103e-02,  3.20829563e-02,  9.95190442e-02,
        -7.24936399e-05, -8.35508108e-05,  3.33697200e-02,
         5.86793497e-02, -1.37900177e-04,  1.61592234e-02,
         9.48483404e-03,  7.21518993e-02, -7.22379627e-06,
         4.75849723e-03,  4.36641686e-02, -3.39368926e-05,
         3.64637934e-02, -6.26850233e-05, -1.82703647e-04,
        -1.88368169e-04,  8.98983479e-02, -5.77150568e-05,
         2.70877965e-02, -9.37035293e-05,  5.68110347e-02,
         5.90063483e-02, -9.77530362e-06, -1.59679155e-04,
         3.17826606e-02,  1.23383127e-01, -5.82275447e-04,
        -4.91398503e-04,  3.94730866e-02, -6.99639204e-05,
        -1.79914059e-06, -4.27950115e-04,  8.35705772e-02,
         3.33169959e-02,  6.20929226e-02, -4.52055130e-04,
         3.77381928e-02, -5.27781376e-04,  6.59131035e-02,
         7.05311745e-02,  3.80479656e-02,  2.85439864e-02,
        -2.71169993e-04,  2.72026639e-02, -2.29235156e-05,
         3.06567829e-03,  6.92931041e-02,  0.00000000e+00,
         0.00000000e+00,  0.00000000e+00,  0.00000000e+00,
         0.00000000e+00,  0.00000000e+00,  0.00000000e+00,
         0.00000000e+00,  0.00000000e+00,  0.00000000e+00,
         0.00000000e+00,  0.00000000e+00,  0.00000000e+00,
         0.00000000e+00,  0.00000000e+00,  0.00000000e+00,
         0.00000000e+00,  0.00000000e+00,  0.00000000e+00,
         0.00000000e+00],
       [-8.66811843e-06,  3.62275690e-02,  5.27299941e-02,
```

```
         1.02231368e-01, -1.55792208e-04, -2.65479321e-05,
        -4.53955909e-05,  3.93020622e-02,  1.43813118e-01,
         5.53143807e-02,  5.14078550e-02, -4.58417344e-05,
        -7.33191555e-05,  5.24425432e-02,  2.23398544e-02,
         7.40078315e-02,  2.26559434e-02,  7.27241263e-02,
         4.25486006e-02,  3.67386565e-02,  6.34456947e-02,
         1.67834740e-02,  5.86199760e-03,  1.54289939e-02,
         1.81464739e-02,  5.42781875e-02,  2.20680702e-02,
         2.18760446e-02,  8.38984847e-02,  3.79876308e-02,
         5.53206801e-02,  8.55532140e-02, -1.32518544e-05,
        -1.53791858e-04, -1.90831546e-04, -1.69591294e-05,
         4.35485542e-02, -1.21442361e-04,  3.46474238e-02,
        -5.84215682e-04, -3.54936696e-04,  4.37195636e-02,
        -7.34442074e-06,  5.90502135e-02,  5.52044250e-02,
        -2.54046063e-06, -2.45426723e-04,  3.98056507e-02,
         8.17805529e-02, -3.74712217e-05,  6.26190379e-02,
         2.30545420e-02,  6.98860288e-02,  7.61621967e-02,
         5.93007430e-02,  1.21739525e-02, -4.43809055e-04,
        -4.93069121e-04, -1.68058505e-05, -1.12943817e-04,
        -1.18279393e-04,  2.17796527e-02, -7.69359758e-07,
        -4.14120586e-06, -1.67652528e-04,  3.05295698e-02,
        -4.56215465e-04,  6.17736438e-03,  3.41870971e-02,
         2.09814403e-02, -3.60745355e-04,  7.75223449e-02,
         5.72856814e-02, -5.16851083e-04,  6.68390170e-02,
        -3.71616916e-05,  3.83356251e-02, -3.95447307e-04,
        -1.68807572e-04, -1.40337739e-04,  0.00000000e+00,
         0.00000000e+00,  0.00000000e+00,  0.00000000e+00,
         0.00000000e+00,  0.00000000e+00,  0.00000000e+00,
         0.00000000e+00,  0.00000000e+00,  0.00000000e+00,
         0.00000000e+00,  0.00000000e+00,  0.00000000e+00,
         0.00000000e+00,  0.00000000e+00,  0.00000000e+00,
         0.00000000e+00,  0.00000000e+00,  0.00000000e+00,
         0.00000000e+00]], dtype=float32), array([0.09689099, 0.1036787 , 0.10248852,
0.10338917, 0.10202771,
       0.09801934, 0.09879171, 0.09902668, 0.0961797 , 0.09702709,
       0.09810887, 0.09642652, 0.09885002, 0.10164196, 0.09164405,
       0.0949537 , 0.1007305 , 0.09801613, 0.1024479 , 0.10862333,
       0.11412376, 0.10210767, 0.09551341, 0.09715071, 0.09484607,
       0.10188241, 0.09590871, 0.09905501, 0.09964921, 0.1023757 ,
       0.1004774 , 0.09701813, 0.10874942, 0.09314054, 0.10119465,
       0.10234515, 0.10749635, 0.10325515, 0.10157112, 0.10771059,
       0.10083356, 0.10073067, 0.09740379, 0.10489494, 0.0994834 ,
       0.10073579, 0.09404648, 0.09387998, 0.10861547, 0.09464745,
       0.10436592, 0.10040105, 0.10213684, 0.09326548, 0.09732251,
       0.09583559, 0.10015965, 0.12356533, 0.08909208, 0.1013594 ,
```

```
       0.10005264, 0.10340626, 0.09468109, 0.09248161, 0.10763872,
       0.1039955 , 0.10536146, 0.09473436, 0.109666  , 0.10082944,
       0.10176305, 0.09515925, 0.09632643, 0.10825914, 0.09972231,
       0.09631392, 0.09540497, 0.10691732, 0.10042662, 0.09705078,
       0.09263724, 0.09212203, 0.09202714, 0.0810589 , 0.08688939,
       0.10520296, 0.09724395, 0.11045086, 0.09668566, 0.06579983,
       0.0803956 , 0.10108782, 0.0885385 , 0.07512747, 0.08445063,
       0.07622636, 0.1318876 , 0.09063533, 0.09848461, 0.07815143],
      dtype=float32)], 'best': {'iter': 10001, 'cost': 0.037655964493751526, 'other_costs':
[0.19405144453048706, 85.92814371257485], 'params': [array([[1.11483477e-01, 1.92378648e-02],
       [3.01854927e-02, 8.42370540e-02],
       [5.98337688e-02, 7.50019699e-02],
       [0.00000000e+00, 2.25062400e-01],
       [1.33209497e-01, 9.81266499e-02],
       [4.08531055e-02, 8.80653411e-03],
       [3.29886898e-02, 6.89789951e-02],
       [3.38914879e-02, 2.15001404e-01],
       [0.00000000e+00, 3.51810694e-01],
       [0.00000000e+00, 3.28007370e-01],
       [2.38535404e-02, 4.79265675e-02],
       [9.44308564e-02, 1.53051736e-02],
       [6.40549511e-02, 1.44989312e-01],
       [3.93945053e-02, 7.07589015e-02],
       [0.00000000e+00, 1.73065767e-01],
       [6.72387853e-02, 3.02629806e-02],
       [7.50776380e-02, 8.97782892e-02],
       [3.45710218e-02, 6.67635873e-02],
       [7.86113814e-02, 1.18180431e-01],
       [9.93348937e-03, 1.52366489e-01],
       [2.78285980e-01, 3.58926244e-02],
       [8.88623148e-02, 4.33318019e-02],
       [3.55960727e-02, 3.96667235e-03],
       [1.00297630e-01, 1.26457717e-02],
       [4.92543739e-04, 1.23981729e-01],
       [8.09945837e-02, 3.98963504e-02],
       [2.23153215e-02, 9.53240395e-02],
       [0.00000000e+00, 3.77375484e-01],
       [5.88899739e-02, 1.27746999e-01],
       [1.12518996e-01, 7.48849437e-02],
       [4.38743159e-02, 4.16703895e-02],
       [6.38468564e-03, 7.11650699e-02],
       [7.59565830e-02, 2.27779061e-01],
       [0.00000000e+00, 3.68534178e-01],
       [1.70470491e-01, 2.34196693e-01],
       [5.83213940e-02, 3.05309352e-02],
```

```
      [7.20524509e-03, 1.49210811e-01],
      [1.57376766e-01, 2.07185894e-01],
      [1.64468847e-02, 1.03404699e-02],
      [3.91270936e-01, 0.00000000e+00],
      [2.47511148e-01, 0.00000000e+00],
      [9.43506211e-02, 9.88454148e-02],
      [1.54151088e-02, 1.09389983e-01],
      [9.04835314e-02, 1.03662580e-01],
      [2.71605421e-02, 2.64651384e-02],
      [9.23164636e-02, 3.42725776e-02],
      [0.00000000e+00, 2.08290666e-01],
      [0.00000000e+00, 2.17942446e-01],
      [0.00000000e+00, 5.18092453e-01],
      [1.84405476e-01, 0.00000000e+00],
      [7.76647180e-02, 2.13509634e-01],
      [6.19360395e-02, 1.75038520e-02],
      [0.00000000e+00, 3.37542921e-01],
      [7.48263747e-02, 1.66284330e-02],
      [7.01982602e-02, 1.00644037e-01],
      [5.34543060e-02, 4.69659679e-02],
      [0.00000000e+00, 3.75528693e-01],
      [3.21293324e-01, 2.53656805e-01],
      [1.83410764e-01, 0.00000000e+00],
      [2.76860118e-01, 7.41430670e-02],
      [1.17539823e-01, 1.93728000e-01],
      [8.01449791e-02, 3.18565071e-02],
      [0.00000000e+00, 2.91028738e-01],
      [0.00000000e+00, 1.05662398e-01],
      [2.16982961e-01, 2.02908725e-01],
      [3.48375030e-02, 4.90755923e-02],
      [2.44770110e-01, 9.53757018e-02],
      [5.48444502e-02, 7.48098716e-02],
      [1.80821478e-01, 2.35227630e-01],
      [3.50163318e-02, 7.94453174e-02],
      [2.27736488e-01, 1.20620303e-01],
      [9.51260105e-02, 2.85057705e-02],
      [6.41506091e-02, 8.07185695e-02],
      [2.70813912e-01, 7.61970207e-02],
      [1.95579343e-02, 5.53474016e-02],
      [2.83693671e-01, 2.18431260e-02],
      [5.32007180e-02, 8.66350308e-02],
      [8.18372741e-02, 3.83403659e-01],
      [1.48443729e-01, 1.22247547e-01],
      [1.46493465e-01, 0.00000000e+00],
      [0.00000000e+00, 0.00000000e+00],
```

```
       [4.41473663e-01, 0.00000000e+00],
       [5.64147711e-01, 0.00000000e+00],
       [0.00000000e+00, 0.00000000e+00],
       [0.00000000e+00, 0.00000000e+00],
       [5.14371216e-01, 0.00000000e+00],
       [6.53186362e-05, 3.49049903e-02],
       [0.00000000e+00, 0.00000000e+00],
       [0.00000000e+00, 0.00000000e+00],
       [0.00000000e+00, 0.00000000e+00],
       [2.66690943e-02, 0.00000000e+00],
       [6.87408924e-01, 0.00000000e+00],
       [0.00000000e+00, 4.80965525e-02],
       [0.00000000e+00, 0.00000000e+00],
       [0.00000000e+00, 0.00000000e+00],
       [0.00000000e+00, 3.38603497e-01],
       [0.00000000e+00, 5.00938714e-01],
       [0.00000000e+00, 0.00000000e+00],
       [0.00000000e+00, 0.00000000e+00],
       [0.00000000e+00, 1.31807569e-02]], dtype=float32), array([[
0.        ,  0.08885651,  0.06356396, ..., -0.25410312,
       -0.78804827, -0.26348507],
       [ 0.06306896,  0.        ,  0.06821403, ..., -0.119032  ,
       -0.25014746, -0.39211074],
       [ 0.10525702,  0.02990693,  0.        , ..., -0.30997866,
       -0.49319315, -0.12219751],
       ...,
       [ 0.02801133,  0.08948383,  0.1000785 , ..., -0.        ,
       -0.3900255 , -0.27609485],
       [ 0.22320949,  0.08815036,  0.05537748, ..., -0.52023107,
       -0.        , -0.5003682 ],
       [ 0.12282094,  0.04667949,  0.04867255, ..., -0.5726365 ,
       -0.1055176 , -0.        ]], dtype=float32), array([[
0.02119593,  0.04002048,  0.03130096,  0.        ,  0.13717611,
        0.        ,  0.01631938,  0.        ,  0.        ,  0.        ,
        0.02094358,  0.01796922,  0.        ,  0.03550774,  0.        ,
        0.02550945,  0.04999774,  0.03026501,  0.03419379,  0.06337688,
        0.07004158,  0.0238702 ,  0.06910674,  0.        ,  0.        ,
        0.08618457,  0.08634388,  0.        ,  0.0625679 ,  0.01234519,
        0.02930681,  0.03208296,  0.09951904,  0.        ,  0.        ,
        0.03336972,  0.05867935,  0.        ,  0.01615922,  0.00948483,
        0.0721519 ,  0.        ,  0.0047585 ,  0.04366417,  0.        ,
        0.03646379,  0.        ,  0.        ,  0.        ,  0.08989835,
        0.        ,  0.0270878 ,  0.        ,  0.05681103,  0.05900635,
        0.        ,  0.        ,  0.03178266,  0.12338313,  0.        ,
        0.        ,  0.03947309,  0.        ,  0.        ,  0.        ,
```

```
        0.08357058,  0.033317  ,  0.06209292,  0.        ,  0.03773819,
        0.        ,  0.0659131 ,  0.07053117,  0.03804797,  0.02854399,
        0.        ,  0.02720266,  0.        ,  0.00306568,  0.0692931 ,
       -0.        , -0.        , -0.        , -0.        , -0.        ,
       -0.        , -0.        , -0.        , -0.        , -0.        ,
       -0.        , -0.        , -0.        , -0.        , -0.        ,
       -0.        , -0.        , -0.        , -0.        , -0.        ],
      [ 0.        ,  0.03622757,  0.05272999,  0.10223137,  0.        ,
        0.        ,  0.        ,  0.03930206,  0.14381312,  0.05531438,
        0.05140786,  0.        ,  0.        ,  0.05244254,  0.02233985,
        0.07400783,  0.02265594,  0.07272413,  0.0425486 ,  0.03673866,
        0.06344569,  0.01678347,  0.005862  ,  0.01542899,  0.01814647,
        0.05427819,  0.02206807,  0.02187604,  0.08389848,  0.03798763,
        0.05532068,  0.08555321,  0.        ,  0.        ,  0.        ,
        0.        ,  0.04354855,  0.        ,  0.03464742,  0.        ,
        0.        ,  0.04371956,  0.        ,  0.05905021,  0.05520443,
        0.        ,  0.        ,  0.03980565,  0.08178055,  0.        ,
        0.06261904,  0.02305454,  0.06988603,  0.0761622 ,  0.05930074,
        0.01217395,  0.        ,  0.        ,  0.        ,  0.        ,
        0.        ,  0.02177965,  0.        ,  0.        ,  0.        ,
        0.03052957,  0.        ,  0.00617736,  0.0341871 ,  0.02098144,
        0.        ,  0.07752234,  0.05728568,  0.        ,  0.06683902,
        0.        ,  0.03833563,  0.        ,  0.        ,  0.        ,
       -0.        , -0.        , -0.        , -0.        , -0.        ,
       -0.        , -0.        , -0.        , -0.        , -0.        ,
       -0.        , -0.        , -0.        , -0.        , -0.        ,
       -0.        , -0.        , -0.        , -0.        , -0.        ]],
      dtype=float32), array([0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.,
0.,
       0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.,
       0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.,
       0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.,
       0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.,
       0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.],
      dtype=float32), array([0., 0.], dtype=float32), array([0.09689099, 0.1036787 ,
0.10248852, 0.10338917, 0.10202771,
       0.09801934, 0.09879171, 0.09902668, 0.0961797 , 0.09702709,
       0.09810887, 0.09642652, 0.09885002, 0.10164196, 0.09164405,
       0.0949537 , 0.1007305 , 0.09801613, 0.1024479 , 0.10862333,
       0.11412376, 0.10210767, 0.09551341, 0.09715071, 0.09484607,
       0.10188241, 0.09590871, 0.09905501, 0.09964921, 0.1023757 ,
       0.1004774 , 0.09701813, 0.10874942, 0.09314054, 0.10119465,
       0.10234515, 0.10749635, 0.10325515, 0.10157112, 0.10771059,
       0.10083356, 0.10073067, 0.09740379, 0.10489494, 0.0994834 ,
       0.10073579, 0.09404648, 0.09387998, 0.10861547, 0.09464745,
```

```
       0.10436592, 0.10040105, 0.10213684, 0.09326548, 0.09732251,
       0.09583559, 0.10015965, 0.12356533, 0.08909208, 0.1013594 ,
       0.10005264, 0.10340626, 0.09468109, 0.09248161, 0.10763872,
       0.1039955 , 0.10536146, 0.09473436, 0.109666  , 0.10082944,
       0.10176305, 0.09515925, 0.09632643, 0.10825914, 0.09972231,
       0.09631392, 0.09540497, 0.10691732, 0.10042662, 0.09705078,
       0.09263724, 0.09212203, 0.09202714, 0.0810589 , 0.08688939,
       0.10520296, 0.09724395, 0.11045086, 0.09668566, 0.06579983,
       0.0803956 , 0.10108782, 0.0885385 , 0.07512747, 0.08445063,
       0.07622636, 0.1318876 , 0.09063533, 0.09848461, 0.07815143],
      dtype=float32)]}, 'costs_history': [(0, [0.18540038168430328, 0.4305814504623413,
49.848637739656915]), (10000, [0.11625630408525467, 0.34096378087997437, 49.85192497532083]),
(20000, [0.09021873772144318, 0.30036434531211853, 52.6579739217653]), (30000,
[0.08822683244943619, 0.2970300316810608, 52.452452452452455]), (40000, [0.08881356567144394,
0.2980160415172577, 52.88270377733598]), (50000, [0.0837930291891098, 0.28947025537490845,
55.7788944723618O6]), (60000, [0.07812200486660004, 0.27950313687324524, 68.44708209693373]),
(70000, [0.07158976048231125, 0.2675626277923584, 72.00791295746785]), (80000,
[0.0628797709941864, 0.2507583796977997, 72.84569138276554]), (90000, [0.056161507964134216,
0.23698419332504272, 76.35619242579324]), (100000, [0.05669838562607765, 0.23811422288417816,
77.9]), (110000, [0.06138213723897934, 0.24775418639183044, 70.84168336673346]), (120000,
[0.04647035524249077, 0.21556983888149261, 81.30653266331659]), (130000, [0.04875117540359497,
0.22079668939113617, 81.1681772406848]), (140000, [0.044512562453746796, 0.21097999811172485,
82.41758241758242]), (150000, [0.04550869017839432, 0.21332766115665436, 81.773399014778321]),
(160000, [0.045827120542526245, 0.21407270431518555, 83.1207065750736]), (170000,
[0.0392603725194931, 0.19814230501651764, 85.55776892430279]), (180000, [0.040708478540182114,
0.20176342129707336, 85.12974051896208]), (190000, [0.041704315692186356, 0.2042163461446762,
85.99605522682445]), (200000, [0.037655964493751526, 0.19405144453048706, 85.92814371257485]),
(0, [0.040507297962903976, 0.20126424729824066, 84.88488488488488])], 'Omega_history':
[(10000, 0.10929739653174557), (20000, 0.047182414491297835), (30000, 0.04072486786615281),
(40000, 0.03496791032644419), (50000, 0.01770106383732387), (60000, 0.016911624082878454),
(70000, 0.012096278980130055), (80000, 0.012761986861794682), (90000, 0.015736177816229352),
(100000, 0.023206092941928918), (110000, 0.019793552065652514), (120000,
0.020880211645097874), (130000, 0.0207613284771259), (140000, 0.029158740803815315), (150000,
0.013748691243640447), (160000, 0.013161904731039273), (170000, 0.012420316211512832),
(180000, 0.013287282179272364), (190000, 0.017553771956492277), (200000,
0.009898570634550968)], 'rng_gradient': RandomState(MT19937) at 0x247C24B9E40,
'rng_validation': RandomState(MT19937) at 0x247C342E040}
```