A parallel implementation of NESTOR in Python 3.4

Andrew Owen Martin

2 October 2017

Contents

1 Introduction

NESTOR is three sets of nodes, retina, memory and matching. The memory nodes are initialised with a “model” pattern, the retina nodes are set with a “target” pattern, and the matching nodes will eventually fire in a way that denotes a position of the model on the retina.

Currently this implementation runs each node on a separate process, which doesn’t challenge the machine much as they all mostly sleep.

2 Canonical description

These are quotes from the paper [6].

> Matching neurons are fully interconnected to all other matching neurons. They are also fully connected to both receptor neurons and memory neurons.

> The output [of matching neurons] will always consist of (parts of) the accepted spike trains.
>
> Matching neurons both maintain and output hypotheses–'where' values–defining possible locations of the target pattern on the retina.
>
> An active matching neuron will output its current hypothesis as a spike train; an inactive matching neuron will adopt a new hypothesis from the first spike train it receives and output this.
>
> Matching neurons evaluate their hypothesis–'where' value–by comparing randomly selected micro-feature(s)–'what' information–from the memory and the corresponding 'what' information from the retina. If the micro-features (the corresponding 'what' values) are the same … the matching neuron becomes active.
>
> An active matching neuron initiates a spike train and retains its current hypothesis (its 'where' value).
>
> A neuron failing to discover the same microfeature will remain inactive and will adopt a new hypothesis encoded in a spike train arriving from other matching neurons.
>
> [I]f the first spike train arrives from memory or retina, a matching neuron will simply output a spike train encoding the position defined by the input trains without further processing of changes to its [notional] state.
>
> A matching neuron only stores the 'where' information … of an accepted spike train.
>
> If a neuron is in an active state, it will select for processing the first spike trains from the retina and memory such that $\Delta^w_{\mathrm{ret}} + \Delta^w_{\mathrm{mem}} = \Delta^w_{\mathrm{neur}}$.
>
> If the comparison is successful (i.e. the 'what' of the memory neuron matches the 'what' of the retina neuron) the matching neuron fires a spike train corresponding to the position defined by its hypothesis Δ^w_{neur}.
>
> Otherwise, the matching neuron will become inactive. In the inactive state the matching neuron will accept information from either the first arriving spike train from another matching neuron or spike trains memory and the retina (albeit with no constraints on ISI's). I think this means it does not wait for a pair of spike trains where the 'where' parts add up to something specific.
>
> If the first spike train comes from a retina and memory cell then it adds up the 'where' parts and emits it. If the first spike comes from a matching cell then it will adopt the 'where' part as its hypothesis and become active.

Other papers that appear to be about NESTOR are [1] [2] [3] [4] [5].

State	Active	Memory	Retina	Retina	Memory	Matching
1	F	F	F	2	3	5
2	F	F	T	-	1	-
3	F	T	F	1	-	-
4	F	T	T	X	X	X
5	T	F	F	6	7	-
6	T	F	T	-	5 or 1	-
7	T	T	F	5 or 1	-	-
8	T	T	T	X	X	X

Table 1: '-' is no change, 'X' is don't care.

3 Finite state machine

See Table 1.

4 Inter-spike Interval (ISI)

The ISI is read-only so a tuple is a suitable data structure, I've given it the `origin` attribute so `Matching`$_{8a}$ neurons can discern where the input came from and hence don't need to have more than one input channel.

4a ⟨*isi class* 4a⟩≡

```
ISI4a = namedtuple('ISI', ('origin', 'position', 'feature'))
```

This code is used in chunk 4b.
Defines:
`ISI`, used in chunks 7c and 10b.

4b ⟨*nestor classes* 4b⟩≡

⟨*isi class* 4a⟩

This definition is continued in chunks 5, 6, 8b, and 14b.
This code is used in chunk 20a.

5 Cell classes

This implementation uses a class hierarchy for cells.

- `Cell`$_{5a}$ → `Matching`$_{8a}$
- `Cell`$_{5a}$ → `DataCell`$_{5c}$ → `Memory`$_{6a}$
- `Cell`$_{5a}$ → `DataCell`$_{5c}$ → `Retina`$_{6c}$

5.1 Cell class

The top cell class just has an `id_num` attribute for convenience when logging.

Cells are expected to have a links attribute defined before they're actually ran. This is a list of other cells to which they will send their ISIs.

5a ⟨*cell class* 5a⟩≡

```
class Cell5a:

    __slots__ = ('id_num','links')

    def __init__5a(self, id_num):
        self.id_num = id_num

    def __repr__5a(self):
        return '{name}#{num}'.format(name=self.name.title(),num=self.id_num)
```

This code is used in chunk 5b.
Defines:
`__init__`, never used.
`__repr__`, never used.
`Cell`, used in chunks 5c and 8a.

5b ⟨*nestor classes* 4b⟩+≡
⟨*cell class* 5a⟩
This code is used in chunk 20a.

5.2 Data cell class

Data cells, i.e. Memory and Retina cells have "where" and "what" attributes.

5c ⟨*data cell class* 5c⟩≡

```
class DataCell5c(Cell5a):

    __slots__ = ('where','what')

    def __init__5a(self, id_num, where, what=None):
        super().__init__5a(id_num)
        self.where = where
        self.what = what
```

This code is used in chunk 5d.
Defines:
`__init__`, never used.
`DataCell`, used in chunks 6 and 7c.
Uses `Cell` 5a.

5d ⟨*nestor classes* 4b⟩+≡
⟨*data cell class* 5c⟩
This code is used in chunk 20a.

5.3 Memory cell

Memory cells are expected to be set at initialisation, and not changed.

6a ⟨*memory cell* 6a⟩≡

```
class Memory6a(DataCell5c):

    name = 'memory'
```

This code is used in chunk 6b.
Defines:
`Memory`, used in chunks 12, 13, and 15.
Uses `DataCell` 5c.

6b ⟨*nestor classes* 4b⟩+≡

⟨*memory cell* 6a⟩

This code is used in chunk 20a.

5.4 Retina cell

Retina cells are expected to be periodically updated with `update_retina`16b.

6c ⟨*retina cell* 6c⟩≡

```
class Retina6c(DataCell5c):

    name = 'retina'
```

This code is used in chunk 6d.
Defines:
`Retina`, used in chunks 12, 13, and 15.
Uses `DataCell` 5c.

6d ⟨*nestor classes* 4b⟩+≡

⟨*retina cell* 6c⟩

This code is used in chunk 20a.

5.5 Wait function

Returns a number of seconds from a bounded Gaussian distribution. Often used with `partial` from the `functools` library.

6e ⟨*wait function* 6e⟩≡

```
def wait6e(mean, sigma, min_wait, max_wait):
    return max(
        min(random.gauss(mean,sigma), max_wait),
        min_wait)
```

This code is used in chunk 7b.
Defines:
`wait`, used in chunks 7c and 9a.

7a ⟨*dependencies* 7a⟩≡

```
import random
```

This definition is continued in chunks 9b and 18–20.
This code is used in chunk 20a.

7b ⟨*nestor methods* 7b⟩≡

⟨*wait function* 6e⟩

This definition is continued in chunks 7d, 10, 11b, 16, 18b, and 19c.
This code is used in chunk 20a.

5.6 Process cell function

This function takes a `DataCell`5c,

7c ⟨*process cell function* 7c⟩≡

```
def process_cell7c(cell, mean, sigma, min_wait, max_wait):
    ''' Take a DataCell5c, and some variables defining firing rate''

    sleep_time = partial(wait6e, mean, sigma, min_wait, max_wait)

    while True:

        next_wait = sleep_time()

        time.sleep(next_wait)

        isi = ISI4a(origin=cell.name, position=cell.where, feature=cell.what)


        for connection in cell.links:

            try:

                connection.receptor.put(isi,block=False)

            except queue.Full:

                pass
```

This code is used in chunk 7d.
Defines:
`process_cell`, used in chunk 17.
Uses `DataCell` 5c, `ISI` 4a, and `wait` 6e.

7d ⟨*nestor methods* 7b⟩+≡

⟨*process cell function* 7c⟩

This code is used in chunk 20a.

5.7 Matching cell

Class definition for the Matching cell. Retina and Memory store data received from their respective channels.

8a ⟨*matching cell* 8a⟩≡

```
class Matching8a(Cell5a):

    name = 'matching'

    __slots__ = ('retina','memory','hypothesis','receptor')

    max_input_isi_count = 1

    def __init__5a(
        self,
        id_num,
        retina=None,
        memory=None,
        hypothesis=None,
        receptor=None,
    ):
        '''id_num = convenience identifier
        Ignore all the other attributes, unless you're copying the
        Matching8a cell as they'll be set during processing.
        receptor will be set during initialisation.'''

        super().__init__5a(id_num)
        self.retina = retina
        self.memory = memory
        self.hypothesis = hypothesis
        if receptor is None:
            receptor = Queue(maxsize=Matching8a.max_input_isi_count)
        self.receptor = receptor
```

This code is used in chunk 8b.
Defines:
`__init__`, never used.
`Matching`, used in chunks 12 and 15.
Uses `Cell` 5a.

8b ⟨*nestor classes* 4b⟩+≡
⟨*matching cell* 8a⟩
This code is used in chunk 20a.

5.8 Process matching cell

9a ⟨*process matching cell* 9a⟩≡

```
def process_matching9a(
    cell,
    mean,
    sigma,
    min_wait,
    max_wait,
    memory_cell_count,
    retina_cell_count):

    not_listening_time = datetime.now()

    sleep_time = partial(wait6e, mean, sigma, min_wait, max_wait)

    process_signal = partial(
        update_matching11a,
        cell,
        memory_cell_count=memory_cell_count,
        retina_cell_count=retina_cell_count,)

    while True:

        signal = cell.receptor.get()

        if False and datetime.now() < not_listening_time:
            continue

        output = process_signal(signal)

        if output:

            matching_fire10b(cell, output)

            not_listening_time = (
                datetime.now()
                + timedelta(0,sleep_time(),),)
```

This code is used in chunk 10a.
Defines:
`process_matching`, used in chunk 17.
Uses `matching_fire` 10b, `update_matching` 11a, and `wait` 6e.

9b ⟨*dependencies* 7a⟩+≡

```
import time
import queue
```

This code is used in chunk 20a.

10a ⟨*nestor methods* 7b⟩+≡
⟨*process matching cell* 9a⟩
This code is used in chunk 20a.

5.9 Matching fire function

10b ⟨*matching fire function* 10b⟩≡

```
def matching_fire10b(cell, output):

    log.info('{cell} fires {output}'.format(cell=cell, output=output))

    isi = ISI4a(origin=cell.name, position=output, feature=None,)

    for connection in cell.links:

        try:

            connection.receptor.put(isi,block=False)

        except queue.Full:

            pass
```

This code is used in chunk 10c.
Defines:
`matching_fire`, used in chunk 9a.
Uses `ISI` 4a.

10c ⟨*nestor methods* 7b⟩+≡
⟨*matching fire function* 10b⟩
This code is used in chunk 20a.

5.10 Update matching cell

11a ⟨*update matching cell* 11a⟩≡

```
def update_matching(cell, signal, memory_cell_count, retina_cell_count):

    fire = None

    if cell.hypothesis is None: # 1,2,3

        ⟨update inactive matching cell 12⟩

        if cell.memory and cell.retina:

            fire = cell.retina + cell.memory

            cell.memory = None
            cell.retina = None

    else: # 5,6,7

        ⟨update active matching cell 13⟩

        if cell.memory and cell.retina:

            if not (cell.memory.feature == cell.retina.feature):

                cell.hypothesis = None

            else:

                fire = cell.hypothesis

            cell.memory = None
            cell.retina = None

    return fire
```

This code is used in chunk 11b.
Defines:
`update_matching`, used in chunk 9a.

11b ⟨*nestor methods* 7b⟩+≡
⟨*update matching cell* 11a⟩
This code is used in chunk 20a.

5.11 Update inactive matching cell

12 ⟨*update inactive matching cell* 12⟩≡

```
if cell.memory is None:

    if cell.retina is None: #1

        if signal.origin == Retina6c.name: # Go from 1 to 2

            cell.retina = signal.position

        elif signal.origin == Memory6a.name: # Go from 1 to 3

            cell.memory = signal.position

        elif signal.origin == Matching8a.name: # Go from 1 to 5

            cell.hypothesis = signal.position

            fire = cell.hypothesis

    elif signal.origin == Memory6a.name: #2

        cell.memory = signal.position # Go from 2 to 1

elif signal.origin == Retina6c.name: #3

    cell.retina = signal.position # Go from 3 to 1
```

This code is used in chunk 11a.
Uses `Matching` 8a, `Memory` 6a, and `Retina` 6c.

5.12 Update active matching cell

13 ⟨*update active matching cell* 13⟩≡

```
if cell.memory is None:

    if cell.retina is None: #5

        if signal.origin == Retina6c.name: # Go from 5 to 6

            if (
                signal.position >= cell.hypothesis
                or (signal.position + memory_cell_count < cell.hypothesis)
            ):

                pass

            elif signal.position < cell.hypothesis:

                cell.retina = signal

        elif signal.origin == Memory6a.name:

            if (
                signal.position >= cell.hypothesis
                or (signal.position + retina_cell_count < cell.hypothesis)
            ):

                cell.hypothesis = None # Go from 5 to 1

            elif signal.position < cell.hypothesis: # Go from 5 to 7

                cell.memory = signal

    elif( #6
        signal.origin == Memory6a.name
        and (cell.retina.position + signal.position == cell.hypothesis)
    ):

        cell.memory = signal # Go from 6 to 5 or 1

        if signal is None:
            raise RuntimeError('signal is none')

elif ( #7
    signal.origin == Retina6c.name
    and (cell.memory.position + signal.position == cell.hypothesis)
):

    cell.retina = signal # Go from 7 to 5 or 1
```

This code is used in chunk 11a.

Uses `Memory` 6a and `Retina` 6c.

6 Network

6.1 Network named tuple

14a ⟨*network class* 14a⟩≡

```
Network14a = namedtuple('Network',('retina','memory','matching'))
```

This code is used in chunk 14b.
Defines:
`Network`, used in chunk 15.

14b ⟨*nestor classes* 4b⟩+≡

⟨*network class* 14a⟩

This code is used in chunk 20a.

6.2 Make Network function

Constructs all the retina cells, then the memory cells, then the matching cells, then links all the data cells to the matching cells and all the matching cells to all the other matching cells.

At the end of this function you get a `Network`$_{14a}$ where all the `Retina`$_{6c}$ cells have `None` as their features, they need to be initialised with `update_retina`$_{16b}$.

15 ⟨*make network function* 15⟩≡

```
def make_network15(retina_count, matching_count, memory_features):

    retina_cells = [
        Retina6c(id_num=retina_num, where=retina_count - retina_num,)
        for retina_num
        in range(retina_count)
    ]

    memory_cells = [
        Memory6a(id_num=memory_num, where=memory_num, what=feature,)
        for memory_num, feature
        in enumerate(memory_features, start=1)
    ]

    matching_cells = [
        Matching8a(id_num=matching_num)
        for matching_num
        in range(matching_count)
    ]

    # Link all data (memory and retina) cells to all matching cells.
    for data_cell in chain(retina_cells, memory_cells):
        data_cell.links = matching_cells

    # Link all matching cells to all other matching cells.
    for num, matching in enumerate(matching_cells):
        others = matching_cells[:num] + matching_cells[num+1:]
        matching.links = others

    return Network14a(
        retina=retina_cells,
        matching=matching_cells,
        memory=memory_cells,)
```

This code is used in chunk 16a.
Defines:
`make_network`, used in chunk 18c.
Uses `Matching` 8a, `Memory` 6a, `Network` 14a, and `Retina` 6c.

16a ⟨*nestor methods* 7b⟩+≡
⟨*make network function* 15⟩
This code is used in chunk 20a.

6.3 Update retina function

Sets the features of the `Retina`$_{6c}$ cells of a `Network`$_{14a}$.

16b ⟨*update retina function* 16b⟩≡

```
def update_retina16b(network, retina_features):
    for cell, feature in zip(network.retina, retina_features):
        cell.what = feature
```

This code is used in chunk 16c.
Defines:
`update_retina`, used in chunk 18c.

16c ⟨*nestor methods* 7b⟩+≡
⟨*update retina function* 16b⟩
This code is used in chunk 20a.

6.4 Run network function

17 ⟨*run network function* 17⟩≡

```
def run_network17(network):

    cell_worker = partial(
        process_cell7c,
        mean=1.5,
        sigma=0.5,
        min_wait=0.1,
        max_wait=10)

    static_cell_processes = [
        Process(target=cell_worker, args=(cell,), daemon=True)
        for cell
        in chain(network.retina, network.memory)
    ]

    matching_worker = partial(
        process_matching9a,
        mean=1.5,
        sigma=0.5,
        min_wait=0.1,
        max_wait=10,
        memory_cell_count=len(network.memory),
        retina_cell_count=len(network.retina),)

    matching_cell_processes = [
        Process(target=matching_worker, args=(cell,), daemon=True)
        for cell
        in network.matching
    ]

    [
        process.start()
        for process
        in chain(static_cell_processes, matching_cell_processes,)
    ]

    # The program doesn't exit until <enter> is pressed.
    input()
```

This code is used in chunk 18b.
Defines:
`run_network`, used in chunk 18c.
Uses `process_cell` 7c and `process_matching` 9a.

18a ⟨*dependencies* 7a⟩+≡
```
from multiprocessing import Process, Queue
from functools import partial
from itertools import chain
```
This code is used in chunk 20a.

18b ⟨*nestor methods* 7b⟩+≡
⟨*run network function* 17⟩
This code is used in chunk 20a.

7 Front end

18c ⟨*front end* 18c⟩≡
```
description = (
    "Run a NESTOR, mostly as a library"
)

if __name__ == '__main__':
    parser = argparse.ArgumentParser(
        description=description,
    )

    # Args go here

    args = parser.parse_args()

    retina_count=100

    retina_features, memory_features = random_task19b(
        retina_count=retina_count,
        memory_count=10,
        noise_count=0,
    )

    network = make_network15(
        retina_count=retina_count,
        matching_count=10,
        memory_features=memory_features,)

    update_retina16b(network=network, retina_features=retina_features,)

    run_network17(network)
```
This code is used in chunk 20a.
Uses `make_network` 15, `random_task` 19b, `run_network` 17, and `update_retina` 16b.

19a ⟨*dependencies* 7a⟩+≡

```
import argparse
```

This code is used in chunk 20a.

7.1 Random task function

19b ⟨*random task function* 19b⟩≡

```
def random_task19b(retina_count, memory_count, noise_count):

    colours = ('black','blue','green','red','yellow','orange',
        'purple','teal','white','brown')

    retina_features = tuple(
        random.choice(colours)
        for _
        in range(retina_count)
    )

    mem_num = max(0,(retina_count//2)-memory_count)

    memory_features = retina_features[mem_num:mem_num+memory_count]

    # Add noise.
    for wrong_num in random.sample(range(memory_count),noise_count):
        wrong_feature = retina_features[wrong_num]
        while wrong_feature == retina_features[wrong_num]:
            wrong_feature = random.choice(colours)
        retina_features = (
            retina_features[:wrong_num]
            + (wrong_feature,)
            + retina_features[wrong_num+1:])

    return retina_features, memory_features
```

This code is used in chunk 19c.
Defines:
`random_task`, used in chunk 18c.

19c ⟨*nestor methods* 7b⟩+≡
⟨*random task function* 19b⟩
This code is used in chunk 20a.

8 Library definition

20a ⟨*nestor.py* 20a⟩≡

```
#!/usr/bin/env python
# -*- coding: utf-8 -*-
⟨dependencies 7a⟩
from collections import namedtuple
from datetime import datetime
from datetime import timedelta
from itertools import chain
import random
import time
⟨init logging 20b⟩
⟨nestor classes 4b⟩
⟨nestor methods 7b⟩
⟨front end 18c⟩
```

Root chunk (not used in this document).

8.1 Logging

20b ⟨*init logging* 20b⟩≡

```
log.basicConfig(level=log.INFO)
```

This code is used in chunk 20a.

20c ⟨*dependencies* 7a⟩+≡

```
import logging as log
```

This code is used in chunk 20a.

References

[1] J M. Bishop. NESTOR: A spiking neuron implementation of sds, 2017.

[2] Kris De Meyer. Explorations in stochastic diffusion search: Soft-and hardware implementations of biologically inspired spiking neuron stochastic diffusion networks. Technical report, Technical Report KDM/JMB/2000, 2000.

[3] Kris De Meyer, John Mark Bishop, and Slawomir J. Nasuto. Attention through self-synchronisation in the spiking neuron Stochastic Diffusion Network. *Consc. and Cogn*, 9(2):81–81, 2000.

[4] Thomas Morey. Parallel implementation of a spiking neuron stochastic diffusion network. Technical report, Cybernetics Dept., Reading University, 1999.

[5] Thomas Morey, Kris De Meyer, Slawomir J. Nasuto, and John Mark Bishop. Implementation of the spiking neuron Stochastic Diffusion Network on parallel hardware. *Consciousness and Cognition*, 2000.

[6] Slawomir J Nasuto, John Mark Bishop, and Kris De Meyer. Communicating neurons: A connectionist spiking neuron implementation of Stochastic Diffusion Search. *Neurocomputing*, 72(4):704–712, 2009.

A Index

B Code Chunks