Neuromorphic_Silicon_Photonics

April 12, 2017

Contents

Chapter 1

Solving ODEs with Photonic Modulator Neurons

This notebook was created to support the publication of an article titled *Neuromorphic Silicon Photonics*, authored by Alexander N. Tait, Ellen Zhou, Thomas Ferreira de Lima, Allie X. Wu, Mitchell A. Nahmias, Bhavin J. Shastri and Paul R. Prucnal, first submitted on 5 Nov 2016.

A copy of this notebook is available at:
https://github.com/lightwave-lab/Neuromorphic_Silicon_Photonics

1.1 Modifications to the nengo project

Nengo is based exclusively on monotonic, non-negative output neuron models. However, its encoding-decoding algorithms should work with other kinds of neuron models. Here, we use the following `FourierSinusoid` class of neurons included in our fork of the nengo project.

1.2 The Lorenz chaotic attractor

In this simulation, we chose to construct a neural network using the neurons defined above to solve a classical chaotic dynamical system named "Lorenz attractor".

The equations are:

$$\dot{x}_0 = \nu(x_1 - x_0)\dot{x}_1 = x_0(\rho - x_2) - x_1\dot{x}_2 = x_0x_1 - \beta x_2$$

Since x_2 is centered around approximately ρ, and since NEF ensembles are usually optimized to represent values within a certain radius of the origin, we substitute $x_2' = x_2 - \rho$, giving these equations:

$$\dot{x}_0 = \nu(x_1 - x_0)\dot{x}_1 = -x_0x_2' - x_1\dot{x}_2' = x_0x_1 - \beta(x_2' + \rho)$$

Refer to the standard example of the Lorenz attractor solver with 2000 neurons in a nengo example. *Note that the last equation for x_2' is typically shown with an error in that example and in other articles from Prof. Eliasmith's group.

1.3 Encoding strategy

From here onwards, we will refer the Lorenz system in its reduced form as as $\vec{x} = f(\vec{x})$, with:

$$\vec{x} = [x_0, x_1, x_2']^T$$

and

$$f(\vec{x}) = \begin{bmatrix} \nu(x_1 - x_0) \\ -x_0 x_2' - x_1 \\ x_0 x_1 - \beta(x_2' + \rho) \end{bmatrix}$$

This is specified to nengo like this...

```
# the ODE to emulate
# The default values for sigma, beta and rho originally used by Lorenz.
# Cf. https://en.wikipedia.org/wiki/Lorenz_system#Analysis
nu = 10
beta = 8.0/3
rho = 28
def feedback(x):
    dx0 = (-nu * x[0] + nu * x[1]) / gamma
    dx1 = (-x[0] * x[2] - x[1]) / gamma
    dx2 = (x[0] * x[1] - beta * (x[2] + rho)) / gamma

    return [dx0 * tau + x[0],
            dx1 * tau + x[1],
            dx2 * tau + x[2]]
```

Hooking it up:

```
# The main ensemble
state = nengo.Ensemble(num_neurons, dimensions=3,
    intercepts=intercepts,
    neuron_type=nengo.neurons.FourierSinusoid(max_overall_rate=max_transmiss
                                              s_pi=s_pi),
    max_rates=max_rates,
    encoders=encoders, radius=60.)

# This special node calls a function every timestep,
# in this case a class method of delay
delay_node = nengo.Node(delay.step, size_in=3, size_out=3)

# Connections from state to delay and back
cdel = nengo.Connection(state, delay_node,
                        function=feedback, synapse=tau)
conn = nengo.Connection(delay_node, state)
```

1.3.1 Neuron model

Using nengo, we instantiate a population of N neurons that are all-to-all interconnected. These neurons are responsible of *representing* the vector $\vec{x}$ at any time t. We consider the state of each neuron as $\vec{s} = [s_i]$ for neuron i. The ODE that models each neuron, in this case, is:

$$\tau s_i + s_i = u_i$$

where u_i represents the post-synaptic input of the neuron and $y_i = \sigma(s_i)$ its output.

1.3.2 Nengo encoding strategy

In order to *encode* a vector $\vec{x}$ in the population N, nengo performs the following linear transformation (it has to be linear for the method to work):

$$s_i = g_i \vec{e}_i \cdot \vec{x} + b_i$$

where g_i is a gain term, $\vec{e}_i$ is an encoder vector, and b_i is a bias term. This is called the *encoding strategy*.

Nonlinear operations are effectively performed by linear combinations of the neural nonlinearities $\sigma(s_i)$. Therefore, it is the encoder's mission to generate as much entropy about the variables $\vec{x}$ as possible. This can be done by generating a diverse set of $(g, \vec{e}, b)$ parameters. Below, we do this by using $\vec{e}_i = [1, \pm 1, \pm 1]$, mixing all components of $\vec{x}$ together. Note: this can be optimized even further by noticing that the ODE does not contain $x_0 x_2$ terms.

Because we know that σ is a sinusoid, we create a set of (g, b) values to span a Fourier-like basis of functions across the domain $\vec{e}_i \cdot \vec{x} \in [-1, 1]$. (See tuning curves).

```
# Intercept, in this case, corresponds to where the tuning curve intercepts
# zero. Range of [-.5, .5] corresponds to [-pi, pi]
ints = [0, 1/4]
# This number represents how many periods do we want between -1 and 1
# (see tuning curves below)
rats = s_pi * np.arange(1, 4)/2
# Encoder multipliers
enst = [-1,1]

num_intercepts = len(ints)
num_max_rates = len(rats)
num_encoders = len(enst) ** 2

j = 0
encoders = np.zeros(shape=(num_neurons, 3))
intercepts = np.zeros(num_neurons)
max_rates = np.zeros_like(intercepts)
for ir in range(num_max_rates):
    for ii in range(num_intercepts):
        for ie0 in range(len(enst)):
            if ie0 is 0:
                continue
            for ie1 in range(len(enst)):
                for ie2 in range(len(enst)):
                    vertex = np.array([enst[ie0], enst[ie1], enst[ie2]])
                    if not np.all(vertex == 0):
                        encoders[j,:] = vertex
                        intercepts[j] = ints[ii]
                        max_rates[j] = rats[ir]
                        j += 1
```

1.3.3 Timing

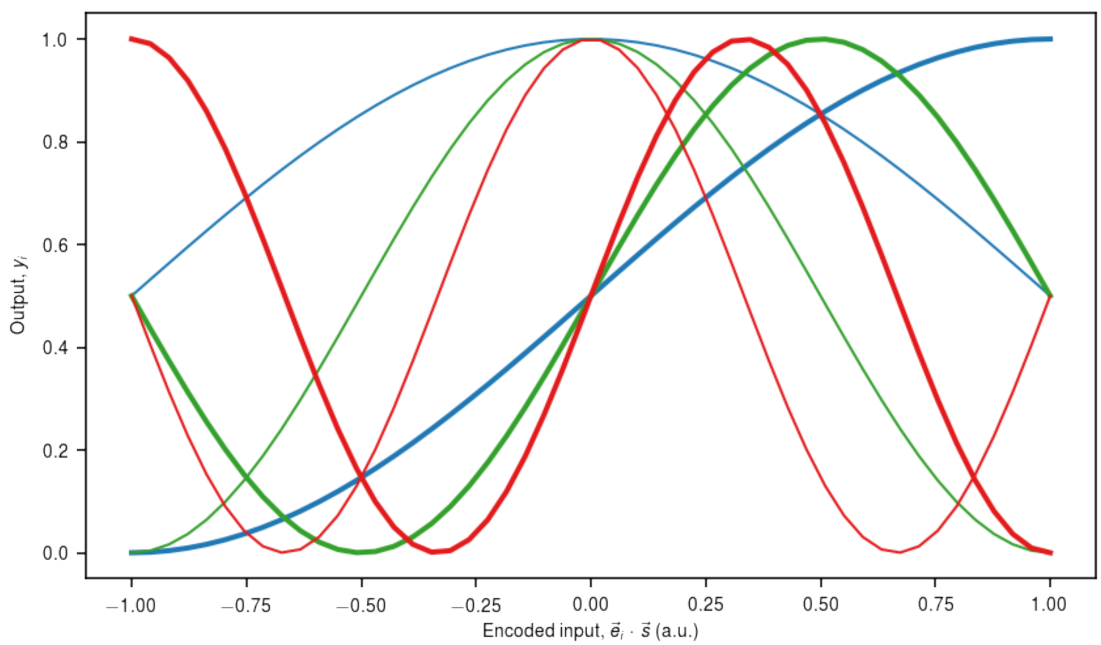
```
# Round-trip feedback delay in ns
delayTime = .048
# gamma is a characteristic time scale in real time units
# The coefficient gamma/delayTime determines the stability
# In paper, coefficient was 65 (spurious), 104 (inaccurate), 260 (looks good
gamma = 260 * delayTime

# We'll make a simple object to implement the delayed feedback
class Delay(object):
```

```
    def __init__(self, dimensions, timesteps=50):
        timesteps = max(timesteps,1)
        self.history = np.zeros((timesteps, dimensions))
    def step(self, t, x):
        self.history = np.roll(self.history, -1, axis=0)
        self.history[-1] = x
        return self.history[0]
delay = Delay(3, timesteps=int(delayTime / dt))
```

1.3.4 Tuning curves in Fourier basis

Here, assuming that the neuron states are $s_i = g_i \vec{e}_i \cdot \vec{x} + b_i$, we plot the functions $\sigma(s_i)$ for neurons with different g_i, b_i values according to the previous table.



1.4 Decoding strategy: calculating weight matrix

As mentioned, nengo decodes a function $h(\vec{x})$ from the population of neurons by a linear decoding strategy, i.e. a matrix $d^{(h)}$ resulting in an estimator $\hat{h}(\vec{x})$:

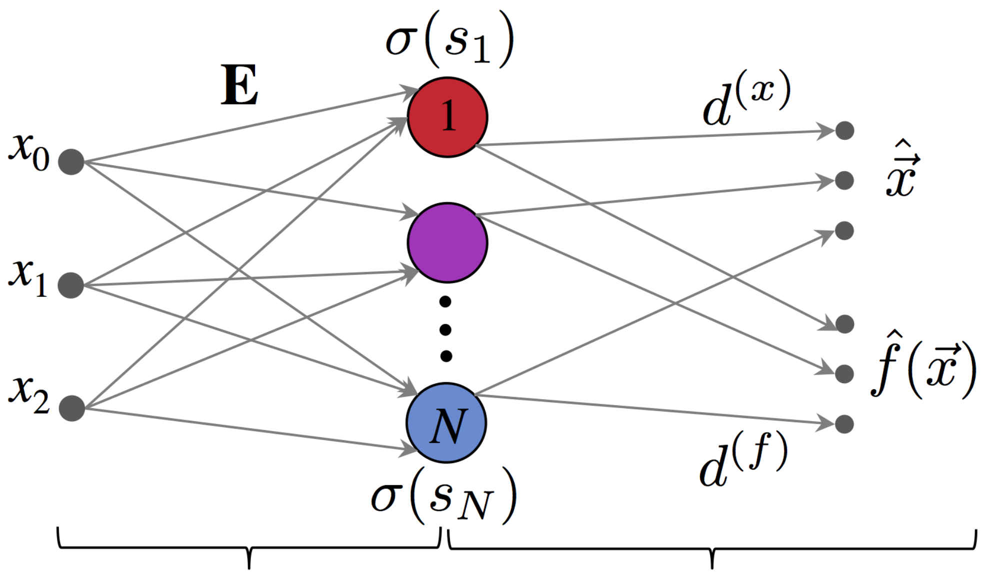
$$\hat{h}(\vec{x}) = d^{(h)} \vec{y}$$

where $y_i = \sigma(s_i) = \sigma(g_i \vec{e}_i \cdot \vec{x} + b_i)$

.

This matrix $d^{(h)}$ is uniquely dependent on the encoder strategy, the neuron's transfer function σ and the function h. As a result, it can be pre-computed before any real-time simulation. Namely, it attempts to minimize the following objective function:

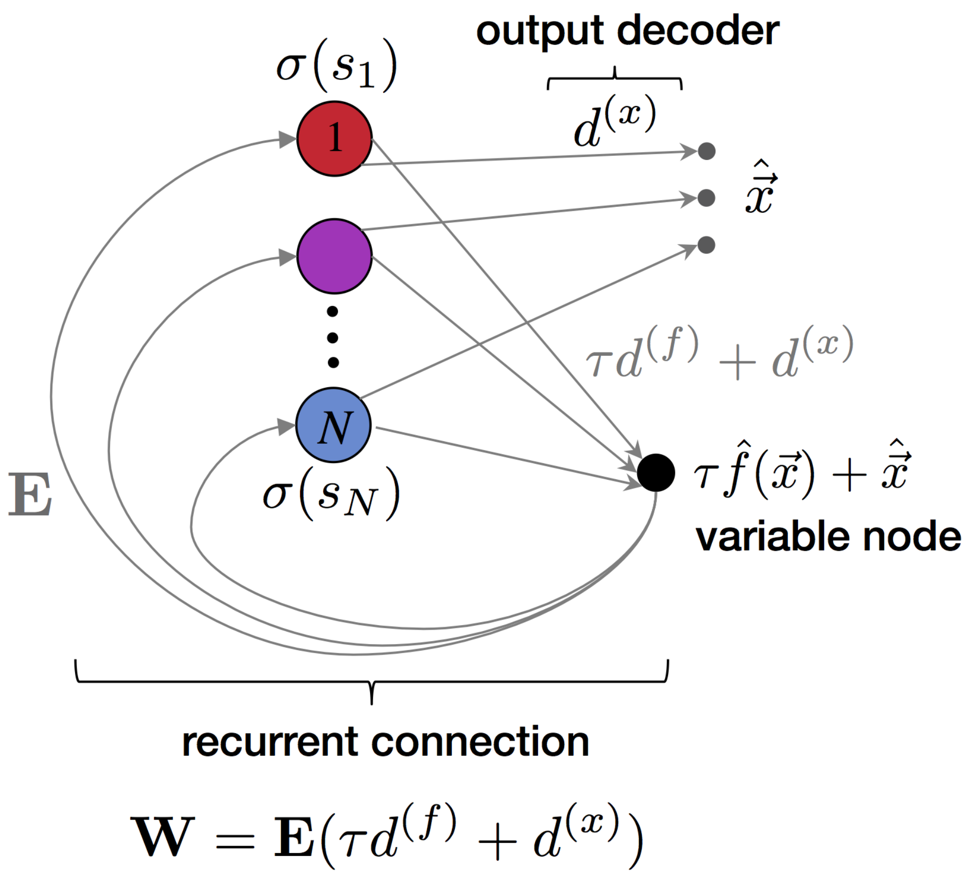
$$J = \int \left\| d^{(h)} \vec{y} - h(\vec{x}) \right\| \mathrm{d}\vec{x}$$



encoding **decoding**

$$s_i = g_i \vec{e_i} \cdot \vec{x} + b_i \qquad \hat{\vec{x}} = \boxed{d^{(x)}} \cdot \sigma(\vec{s})$$

$$\Rightarrow \vec{s} = \mathbf{E} \cdot \vec{x} + \vec{b} \qquad \hat{f}(\vec{x}) = \boxed{d^{(f)}} \cdot \sigma(\vec{s})$$



where the integral is over the desired range of values of $\vec{x}$.

The minimum can be calculated via the Moore-Penrose pseudoinverse method (Stewart et al. Front Neuroinform. 3 (2009)):

$$\Gamma_{ij} = \int y_i y_j \mathrm{d}\vec{x}$$

$$\Upsilon_i = \int y_i h(\vec{x}) \mathrm{d}\vec{x}$$

$$d^{(h)} = \Gamma^{-1} \cdot \Upsilon$$

1.4.1 Weight matrix

If we add an all-to-all recurrent connection to the neural population, their collective dynamics is described by the following ODE system:

$$\tau \dot{\vec{s}} + \vec{s} = \overline{\overline{W}} \sigma(\vec{s}) + \vec{I}$$

where $\overline{\overline{W}}$ is the weight matrix and $\vec{I}$ a bias vector.

Nengo sets $\overline{\overline{W}} = \overline{\overline{E}}(d^{(x)} + \tau d^{(f)})$ and $\vec{I} = \vec{b}$, where $\overline{\overline{E}}_{ij} = (\vec{e}_i)_j$. When applied to the ODE above, it is easy to see that one can recover the Lorenz system:

$$\overline{\overline{E}}(\tau\dot{\vec{x}} + \vec{x}) = \overline{\overline{E}}(\hat{\vec{x}} + \tau\hat{f}(\vec{x}))$$

$$\implies \dot{\vec{x}} = f(\vec{x}) + \epsilon(\vec{x})$$

where $\epsilon(\vec{x}) = (1/\tau)(\hat{\vec{x}} - \vec{x}) + \hat{f}(\vec{x}) - f(\vec{x})$.

Below, we show the computed weight matrix $\overline{\overline{W}}$ for this system.