SYDE 556/750 --- Assignment 5

Due Date: Dec 23, 2021

Student ID: 00000000

Note: Please include your numerical student ID only, do *not* include your name.

Note: Unlike assignments 1-4, for this assignment the full instructions (including some hints) are in this file. The cells you need to fill out are marked with a "writing hand" symbol. Of course, you can add new cells in between the instructions, but please leave the instructions intact to facilitate marking.

- This assignment is worth 30 marks (30% of the final grade). The number of marks for each question is indicated in brackets to the left of each question.
- Clearly label any plot you produce, including the axes. Provide a legend if there are multiple lines in the same plot.
- You won't be judged on the quality of your code.
- All questions use the nengo default of Leaky Integrate-and-Fire neurons with the default parameter settings (`tau_rc=0.02` and `tau_ref=0.002`).
- Make sure to execute the Jupyter command "Restart Kernel and Run All Cells" before submitting your solutions. You will lose marks if your code fails to run or produces results that differ significantly from what you've submitted.
- Rename the completed notebook to `syde556_assignment_05_<STUDENT ID>.ipynb` and submit it via LEARN. The deadline is at 23:59 EST on Dec 23, 2021.
- Due to the fact that we have to submit the final grades for the course a few days after the deadline, extensions will only be granted if there are significant extenuating circumstances. Please contact terry.stewart@gmail.com if this is needed.

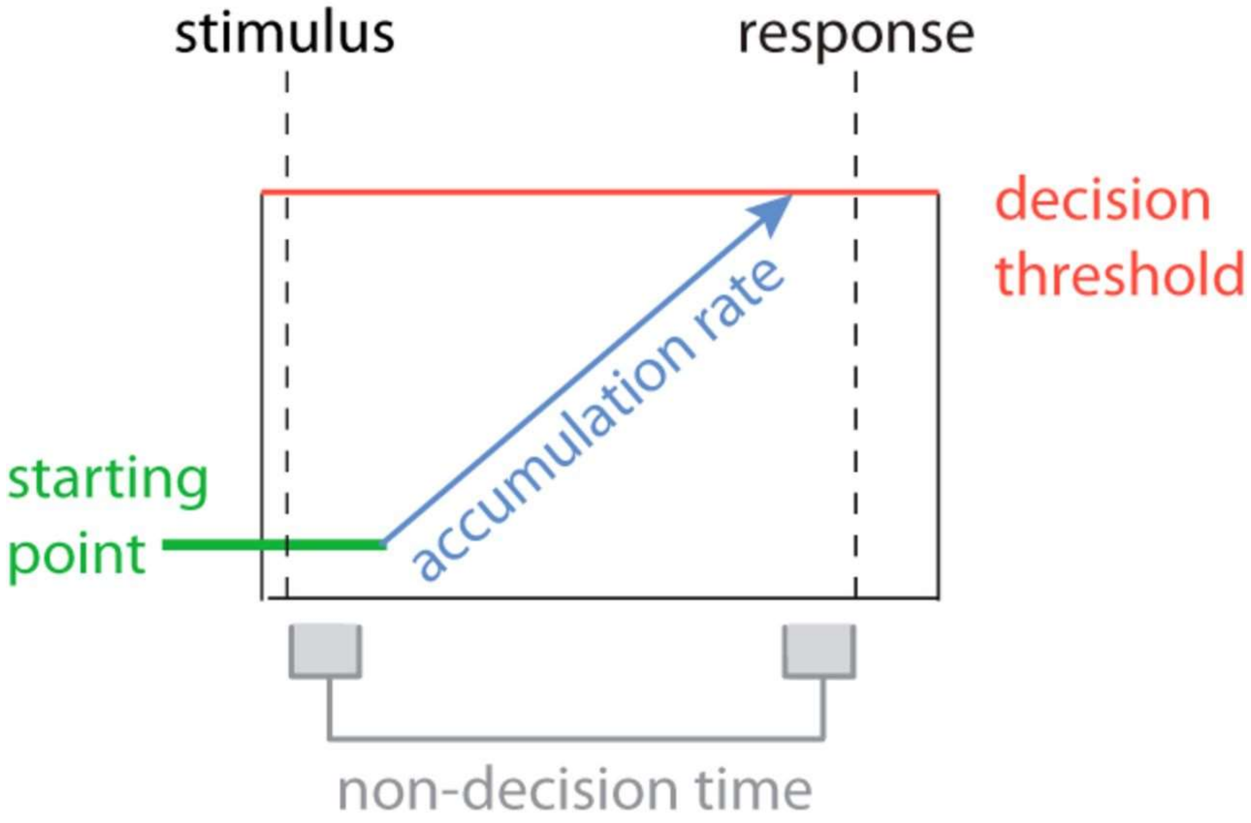
In [ ]:

```
# Import numpy and matplotlib
import numpy as np
import matplotlib.pyplot as plt

import nengo

# Fix the numpy random seed for reproducible results
np.random.seed(18945)

# Some formating options
%config InlineBackend.figure_formats = ['svg']
```

1. Building an Accumulate-to-Threshold Decision Making Model

One standard account for how brains make simple decision-making tasks is that they gradually accumulate evidence for or against something, and when that evidence hits some threshold, a decision is made. This sort of model is used to account for the fact that people take longer to make decisions when the evidence is weak.

If you want more background on this, https://www.jneurosci.org/content/34/42/13870 gives a decent overview, but this diagram shows a high-level overview:



We're going to make a model of this process. It will make its choice based on a single input value, which gives some evidence as to which choice should be made. It will indicate a choice by outputting either a 1 or a -1. If that input evidence is positive, it will be more likely to make the first choice (outputting a 1), and if the input evidence is negative it will be more likely to make the second choice (outputting a -1).

TIP: The Nengo GUI built-in tutorials 10 through 18 may be useful to give you an overview of different recurrent systems and different ways of modifying `Ensembles`.

a) Accumulation. [2 marks] Start by building a recurrent system that can add up evidence over time (the accumulator or integrator). This is a neural `Ensemble` that holds a single dimension, and uses a small number of neurons (50). Provide it with one input `Node` that has a constant value of `[0.1]` and connect that input into the `Ensemble` with a `Connection`. Now make a `Connection` from the `Ensemble` back to itself that computes the identity function. Since this

`Connection` is accumulating evidence over time, we want it to be fairly stable, so set `synapse=0.1` on this `Connection` (leave the other `Connection` at its default value). This means that the neurotransmitter being used will spread out over 100ms, rather than the default 5ms.

If you run the above system with the constant positive input of 0.1 as noted above, the value stored in the accumulator should gradually increase until it hits 1 (this should take about 1 second of simulated time). If you change the input to be -0.1, it should gradually decrease until it hits -1.

Make a single plot that shows the behaviour of the model for four different inputs: 0.2, 0.1, -0.1, and -0.2. For each input, run the model for 2 seconds (`sim.run(2)`) and plot the value stored in the accumulator `Ensemble`. Use a `Probe` synapse of 0.01 to get the stored value.

In [24]:
```
# ✍ <YOUR SOLUTION HERE>
```

b) Accumulator Discussion. [1 mark] What is the mathematical computation being performed here (i.e. what is the relationship between the input and the output)? Why does the value stop increasing (or decreasing) when it hits +1 (or -1)?

✍ \<YOUR SOLUTION HERE>

c) Adding random noise to the neurons. [1 mark] Next, we can add randomness to the neurons. In standard (non-neural) accumulator models, there is a "random-walk" component that randomly varies the value being accumulated. We can model this by adding random noise into the `Ensemble`, which means adding random current to each of the neurons. The command for this is:

```
acc.noise = nengo.processes.WhiteSignal(period=10, high=100, rms=1)
```

(where `acc` is whatever name you gave your accumulator `Ensemble`.)

The strength of this noise is set by the `rms=1` parameter. Generate the same plot as in part (a) but with the noise `rms=1`. Also generate the same plot for `rms=3`, `rms=5`, and `rms=10`. What happens to the resulting output?

In [5]:
```
# ✍ <YOUR SOLUTION HERE>
```

✍ \<YOUR SOLUTION HERE>

e) Adding decision-making. [2 marks] To complete the basic model, we want to determine when this accumulator passes some threshold. If the value becomes large enough, we should make one choice (+1), and if it becomes small enough we should make the other choice (-1). To achieve this, make a new output `Ensemble` that is also one-dimensional and has 50 neurons. Form a `Connection` from the accumulator to this new `Ensemble` that computes the following function:

```
def choice(x):
    if x[0] > 0.9:
        return 1
    elif x[0] < -0.9:
        return -1
    else:
        return 0
```

This new output should now stay at zero until the accumulator value gets large enough, and then quickly move to +1 or -1.

Build this model and plot the output of both the accumulator `Ensemble` and the decision-making `Ensemble`. Use a noise `rms=3` and for both `Probe`s use a synapse of 0.01. Do this for all four input values (0.2, 0.1, -0.1, and -0.2).

How well does the system perform? Does it make decisions faster when there is stronger evidence? What differences are there (if any) between the computation we are asking the system to perform and the actual result?

TIP: try running the model a few times to see the variability in the output

In [28]:
```
# ✍ <YOUR SOLUTION HERE>
```

✍ \<YOUR SOLUTION HERE>

f) Combining Ensembles. [2 marks] An alternative implementation would be to combine the two separate 1-dimensional `Ensembles` into one 2-dimensional `Ensemble`. The Connections are made similarly as in the original model, but they need to target the particular dimensions involved using the `ens[0]` and `ens[1]` syntax. Try building the model this way and plot the results. Do this for a single `Ensemble` with 100 neurons (the same number as the total number of neurons in the original model) and with 500 neurons. Also, be sure to increase the `radius` as would be appropriate in order to produce values like what we had in the original model, where the accumulator might be storing a 1 and the output might be a 1.

How does combining Ensembles in this way change the performance of the system?

When the Ensembles are combined together in this way, what are we changing about the biological claims about the model? In particular, how might we determine whether the real biologicial system has these as separate `Ensembles` or combined together?

In [30]:
```
# ✍ <YOUR SOLUTION HERE>
```

✍ \<YOUR SOLUTION HERE>

g) Improving Representation [2 marks]. Returning to the original implementation from section (e) (with 2 separate Ensembles), we can improve the performance by adjusting the tuning curves of the second `Ensemble`. Do this by setting `intercepts = nengo.dists.Uniform(0.4, 0.9)`. This randomly chooses the x-intercepts of the neurons uniformly between 0.4 and 0.9, rather than the default of -1 to 1. Generate the same plot as in part (e).

How does this affect the performance of the model? (Try running the model a few times to see the variability in performance).

Why does the output stay at exactly zero up until the decision is made (rather than being randomly jittering around zero, as in the previous models)?

Why couldn't we use this approach in the case from part (f) where the `Ensembles` are combined?

In [11]:
```
# ✍ <YOUR SOLUTION HERE>
```

✍ \<YOUR SOLUTION HERE>

2. Temporal Representation

In class, we discussed the Legendre Memory Unit (LMU), a method for storing input information over time. This allows us to make connections where the function being computed is a function of the input over some window in time, rather having to be a function of the current input.

In this question, we will use this to build a model that can distinguish a 1Hz sine wave from a 2Hz sine wave. Notice that it is impossible to perform this task without having information over time; if I just give you a single number at any given point in time, you can't tell whether it's from a 1Hz sine wave or a 2Hz sine wave. So we need some method to store the previous input information, and that's what the LMU does.

a) Representing Information over Time. [2 marks] The core of the LMU is to compute the differential equation $\frac{dx}{dt} = Ax + Bu$ where A and B are carefully chosen using the following math:

```
A = np.zeros((q, q))
B = np.zeros((q, 1))
for i in range(q):
    B[i] = (-1.)**i * (2*i+1)
    for j in range(q):
        A[i,j] = (2*i+1)*(-1 if i<j else (-1.)**(i-j+1))
A = A / theta
B = B / theta
```

Implement this in Nengo. Use `theta=0.5` and `q=6`. The model should consist of a single `Ensemble` that is `q`-dimensional. Use 1000 neurons in this `Ensemble`. Use `synapse=0.1` on both the recurrent `Connection` and on the input `Connection`.

For the input, give a 1Hz sine wave for the first 2 seconds, and a 2Hz sine wave for the second 2 seconds. This can be done with:

```
stim = nengo.Node(lambda t: np.sin(2*np.pi*t) if t<2 else np.sin(2*np.pi*t*2))
```

Run the simulation for 4 seconds. Plot `x` over the 4 seconds using a `Probe` with `synapse=0.01`. `x` should be 6-dimensional, and there should be a noticable change between its value before `t=2` and after `t=2`.

b) Computing the function. [2 marks] We now want to compute our desired function, which is "output a 1 if we have a 1Hz sine wave and a 0 if we have a 2Hz sine wave". To do this, we need to make a `Connection` from the LMU `Ensemble` out to a new `Ensemble` that will be our category. Have it be 1-dimensional with 50 neurons.

Normally in Nengo, when we define a `Connection` we specify a Python function that we want to approximate. Nengo will then choose a bunch of random `x` values, call the function to determine what the output should be for each one, and use that to solve for the decoders. However, in this case, we already have that set of `x` values! That's exactly the data you plotted in part (a). For the x values from t=0 to t=2.0 we want an output of 1. For the x values from t=2.0 to t=4.0, we want an output of -1. So, to specify these target values, we make a matrix of size `(4000,1)` (4000 for the 4000 time steps that you have x values for, and 1 for the output being 1-dimensional). Set the first 2000 values to 1 and the second 2000 values to -1.

Now that you have your `x` values and the corresponding `target` values, you can tell Nengo to use them when you make the `Connection` like this:

```
nengo.Connection(a, b, eval_points=x_values, function=target)
```

That will tell Nengo just to use the values you're giving it, rather than randomly sampling `x` and calling a function to get the target values.

Build this model and plot the resulting category (with a `Probe` with `synapse=0.01`). The output should be near 1 for the first 2 seconds, and near -1 for the second 2 seconds. (Important note: it will not be perfect at this task!)

In [110…
```
# ✍ <YOUR SOLUTION HERE>
```

c) Adjusting the input. [2 marks] Repeat part b) but with an input that is a 2Hz sine wave for the first 2 seconds, and a 1Hz sine wave for the second 2 seconds (i.e. the opposite order as in part (b)). How well does this perform? Describe the similarities and differences. One particular difference you should notice is that the model may make the wrong classification for the first 0.25 seconds. Why is this happening? What could you change to fix this?

In [111…
```
# ✍ <YOUR SOLUTION HERE>
```

✍ \<YOUR SOLUTION HERE>

d) Adjusting the number of neurons. [2 marks] Repeat part b) but adjust the number of neurons in the `Ensemble` computing the differential equation. Try 50, 100, 200, 500, 1000, 2000, and 5000. How does the model behaviour change? Why does this happen? In addition to looking at the actual results for each run, also plot the RMSE in the classification as you adjust the number of neurons.

In [112…
```
# ✍ <YOUR SOLUTION HERE>
```

✍ \<YOUR SOLUTION HERE>

e) Adjusting the q value. [2 marks] Repeat part b) (returning to 1000 neurons) but adjust the value of `q` . Try 1, 2, 4, 8, 16, 32, and 64. How does the model behaviour change? Why does this happen? In addition to looking at the actual results for each run, also plot the RMSE in the classification as you adjust the number of neurons.

In [13]:
```
# ✍ <YOUR SOLUTION HERE>
```

✍ \<YOUR SOLUTION HERE>

3. Online Learning

Normally when build models with the Neural Engineering Framework, we compute the connection weights at the beginning and then leave them fixed while running the model. But, we can also apply online learning rules to adjust the connection weights over time. This has the effect of changing the function being computed. One general learning rule is the PES rule, where you provide an extra input that indicates whether the output value should be increased or decreased. This is generally called an error signal.

a) Basic online learning. [2 marks] Build a network that will learn the identity function. You will need three `Ensembles`, one for the input, one for the output, and one for the error. Each one is 1-dimensional and uses 200 neurons. For the input, use Nengo to randomly generate a 2Hz band-limited white noise signal as follows:

```
stim = nengo.Node(nengo.processes.WhiteSignal(period=100, high=2, rms=0.3))
```

When making the learning connection, initialize it to compute the zero function and to use the PES learning rule as follows:

```
def initialization(x):
    return 0
c = nengo.Connection(pre, post, function=initialization,
learning_rule_type=nengo.PES(learning_rate=1e-4))
```

The error `Ensemble` should compute the difference between the output value and the desired output value. For this initial question, we want the output value to be the same as the input value (i.e. we are learning the identity function). Then connect the error `Ensemble` to the learning rule as follows:

```
nengo.Connection(error, c.learning_rule)
```

(Note: for this question, leave the `synapse` values on the `Connections` at their default values)

Run the model for 10 seconds and plot the input value and the resulting output value (using a `Probe` with `synapse=0.01`). The output should match the input fairly well after the first few seconds.

```
In [14]: # ✍ <YOUR SOLUTION HERE>
```

b) Error calculation. [1 mark] What would happen if you reversed the sign of the error calculation (i.e. if you did `target - output` rather than `output - target`? Why does that happen?

✍ \<YOUR SOLUTION HERE>

c) Computing metrics. [1 mark] Break your data up into 2-second chunks and compute the Root-Mean-Squared-Error between the target value (the stimulus itself) and the output from the model for each chunk. Since the simulation is 10 seconds long, you should have 5 RMSE measures (one for the first 2 seconds, one for the second 2 seconds, one for the third 2 seconds, and so on). Repeat the simulation 10 times and plot the average for each of these values. The result should show that the model gets better over time, but does not reach 0 error.

```
In [14]: # ✍ <YOUR SOLUTION HERE>
```

d) Increasing learning time. [2 marks] Repeat part (c), but run the model for 100 seconds instead of 10 seconds. How do the results change?

In [14]:
```
# ✍ <YOUR SOLUTION HERE>
```

✍ \<YOUR SOLUTION HERE>

e) Learning rates. [2 marks] Repeat part (d), but decrease the learning rate to `1e-5`. How do the results change? How do they compare to part (c)?

In [14]:
```
# ✍ <YOUR SOLUTION HERE>
```

✍ \<YOUR SOLUTION HERE>

f) Improving performance. [1 mark] If you wanted to make the learned result even more accurate, how would you do this? What would you change about the model and learning process?

✍ \<YOUR SOLUTION HERE>

g) Learning other functions. [1 mark] Repeat part (a), but have the system learn a function where the input is a scalar x, but the output is the vector $[x^2, -x]$. This will involve changing the dimensionality of some of the `Ensembles` and adding a `function=` to be computed on the `Connection` from the `stim` to the `error`.

In [19]:
```
# ✍ <YOUR SOLUTION HERE>
```