Assignment 7: Learning connection weights

This assignment should be done in pairs.

Goal

The goal of this assignment is to model supervised learning in Nengo. You will first explore connection weights more generally, and then build a model that learns connection weights between two ensembles, instead of calculating optimal weights like we did up to now.

Assignment

- Download the attached file to use as your starting model. You will see a stimulus node `stim`, a `pre` and a `post` ensemble. Here, the connection weights are calculated as before, and if you run the model, you will see that `post` indeed represents the same value as `pre` (and `stim`).

While pre-calculating optimal connection weights is very efficient and allows us to develop large scale models, it does not explain how these connection weights were learned in the first place. In this assignment, you will look at how Nengo can learn such weights.

To start:

- add two ensembles called `pre_learning` and `post_learning`, each having 10 neurons and representing 1 dimension.
- connect `stim` to `pre_learning`. (`stim` is now connected to both `pre` and `pre_learning`).

Question 1: Exploring weights

First, let's have a look at the weights that Nengo calculates for us:

- connect `pre_learning` to `post_learning` with the following code:

```
learning_connection = nengo.Connection(pre_learning, post_learning,
solver=nengo.solvers.LstsqL2(weights=True))
```

- add the following code to print the weights:

```
#build model without actually running it
with nengo.simulator.Simulator(model, progress_bar=False) as weights_sim:
    pass

#get weights and print
weights = weights_sim.data[learning_connection].weights
print(weights)
```

The code above first builds the model without actually running it, in order to make Nengo calculate the weights. It then takes the weights from the simulation and prints them.

Question 1a: why do we get a 10x10 matrix? What do the numbers represent?

We can also set the weights directly, if we make a connection from an ensemble's *neurons* to another ensemble's *neurons* (note that connections are normally defined at the level of *represented values*, not of the *neurons*). Change the connection that you made above to the following one:

```
learning_connection = nengo.Connection(pre_learning.neurons, post_learning.neurons,
transform=np.random.random(size=(10,10))*2-1)
```

This connects the neurons from `pre_learning` to the neurons of `post_learning`, using random weights between -1 and 1.

Question 1b: open plots for the values of `pre_learning` and `post_learning`, and run this model. Reset the model by clicking the reverse button. Run the model again. Can you see a relation between the values of both ensembles for different runs? Explain why there is a relation or not.

Now connect the first neuron of `pre_learning` to the first neuron of `post_learning`, the second neuron to the second one, etc., by using the identity matrix as transform argument: `np.identity(`*…n_neurons…*`)` in the previous connection.

Question 1c: run the model. Explain why `post_learning` does not represent the same value as `pre_learning`, even though the neurons are directly connected.

Question 1d: what would be the requirement for the above to work?
Try out if this works by setting the seed argument of both ensembles (the seed argument sets the seed of the random number generator).

Question 2: Learning weights

To learn the weights we will use PES learning – prescribed error sensitivity – a form of supervised learning.

Replace the connection that you made above with the following:

```
learning_connection = nengo.Connection(pre_learning, post_learning, transform=0,
learning_rule_type=nengo.PES(learning_rate=1e-5))
```

This makes a connection with a transform to 0 that uses PES learning.

Question 2a: What do you think the connection weights will be in this case?
You can check using the code above.

Because PES learning is a form of supervised learning – that is, we tell the learning algorithm what the outcome should be, or rather, what the current error is – we have to represent the current error and connect that to the learning rule.

- Add a new 10-neuron ensemble called `error`.
- Connect it to the learning rule with:

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

- Now think about how you can calculate the error between the value you want in `post_learning` and the value you currently have. (Hint: you need to make connections from `pre_learning` and `post_learning` to `error` and specify transforms).

- If you run the model, it should now learn to represent the value of `pre_learning` in `post_learning`.

Question 2b: I set the learning rate to a rather low value, so you can observe the learning (the default value is 1e-4). Try what happens if you set the learning rate to .01 or .1. Can you think of an explanation?

- Finally, make both connections (`pre` -> `post` and `pre_learning` -> `post_learning`) represent or learn to calculate the square. You might want to set the learning rate to 1e-4 for this.

Question 2c: We now developed a model that learns the connection weights between two ensembles. However, to do so, we had to specify the error: basically reintroducing pre-calculated connection weights. Think about how this would work when doing, for example, alphabet-arithmetic. Where does the error signal come from in that case? What are therefore the requirements for using supervised learning?

What to hand in:

- A pdf with the answers to the questions.
- The python file with your final model, learning to represent the square between `pre_learning` and `post_learning`.