Simple Data Input and Output with Spinnaker - Lab Manual

Introduction

This manual will introduce you to the basics of live retrieval and injection of data (in the form of spikes) for PyNN scripts that are running on SpiNNaker neuromorphic hardware.

PyNN Support

This section discusses the standard support from PyNN related to spike injection and retrieval.

Output

The standard support for data output for a platform such as SpiNNaker, through the PyNN language, is to use the *record* method to declare the need to record, and the *get_data* method, for retrieval of the specific data. In the current implementation of sPyNNaker, all of the data declared to be recorded via record() is stored on the SDRAM of the chips that the corresponding populations were placed on. By writing the data to SDRAM, the data is stored locally and therefore is guaranteed to be read at some point in the future. In the current implementation, if the memory requirements for recording cannot be met, the model will be run for less time, paused whilst the data is extracted, and then resumed. This may be repeated a number of times until the whole simulation has completed.

The issue with the get functions are that they are called after run() completes, and therefore are not live, and so not able to interact with an external device running in real-time. When used with an external simulation, it is possible to call run a number times, extracting the data between each run and passing it to an external simulation. This mode of operation will not work if the external device or simulation cannot also be paused.

Input

The standard support for data input for a platform such as SpiNNaker, through the PyNN language, is to use the neural models SpikeSourceArray and SpikeSourcePoisson. The issue with both of these models is that they are either random rate based (the SpikeSourcePoisson) or have to be supplied in advance with all the spikes to be sent (SpikeSourceArray). As with the output of spikes, it is possible to change the input spikes of a SpikeSourceArray, or the rate of the SpikeSourcePoisson between successive calls to run(). Again, this will only work if the external device or simulation can be paused.

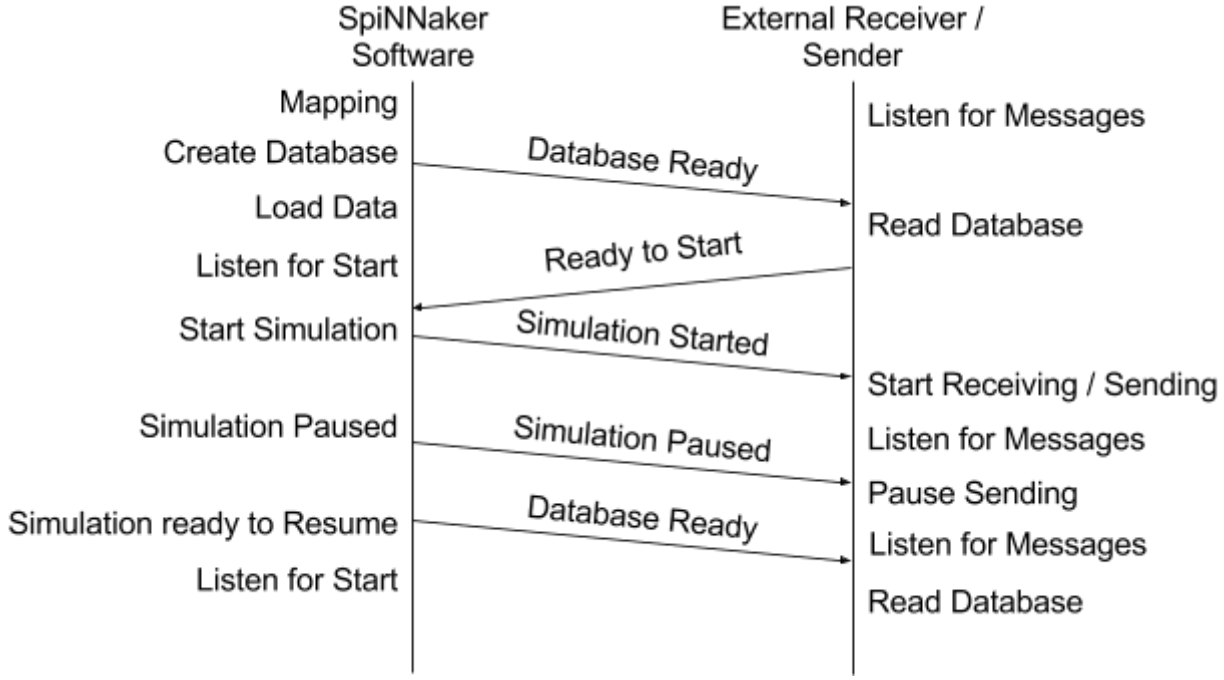
Live I/O Support

PyNN currently doesn't have any support for live interaction with simulations. It is worth noting that future releases of PyNN may use the MUSIC interface to support live injection and retrieval of spikes, but this has not yet been integrated into our software. To compensate for this, sPyNNaker includes an external_devices module that contains support for live injection and retrieval of spikes during a running PyNN simulation, whilst still maintaining the real-time operation of the simulation.

Live Event Database

When live input and output are used in the simulation, a database of the network is created with additional data, such as which SpiNNaker multicast keys are assigned to which neurons. This database can then be used by external receivers and/or senders to work out how to work out which neurons have spiked given the keys received as live output from SpiNNaker, or which keys to inject into SpiNNaker to send a spike into a simulation from outside.

During the mapping process within the software, the database is created, and a message is sent to the external receivers and/or senders that may want to read this database, to let them know that it is ready to be read. In addition, the software will then listen to be told that the database has been read, and that the external receiver or sender is ready for the simulation to start. Just after starting the simulation, the external receivers and senders are again notified so that they can start their activity. This helps to keep these systems in synchronization with the simulation.



Once the simulation has completed, it will pause. At this point the software will send a message to the senders and receivers that the simulation has paused or stopped (it isn't specified which at this point). The receiver or sender can then wait for a new cycle of messages, again starting with the message indicating that the database is ready. This will allow for any changes that have been made to the simulation. The process then goes around the loop again, from database ready, to notify ready to start, to start simulation to pause simulation and so on until all cycles are complete.

Live Output

To activate live retrieval from a given population, the command

p.external_devices.activate_live_output_for(<Population_object>**)**

is used (assuming pyNN.spiNNaker has been imported as p). This informs the sPyNNaker backend to send data out of the SpiNNaker system during the execution of the simulation. Note that this uses an additional core on the board.

Other optional parameters for the **activate_live_output_for()** function are defined below:

Parameter	**Description**
port	The UDP port number to which the SpiNNaker machine will send packets. By default, the port specified as live_spike_port in the [Recording] section of your .spynnaker.cfg file will be used (17895 by default).
host	The host to which the SpiNNaker machine will send packets. By default, the IP address of the machine executing the script is used, but this can be changed if the spikes are to be sent to a different machine.

Live Input

To activate the live injection functionality, you need to instantiate a new *SpikeInjector* neural model is used. The SpikeInjector is considered to be a similar to a SpikeSourceArray, so you can build a population with a number of neurons etc in the normal way, as shown below:

```
injector_forward = Frontend.Population(
       5, p.external_devices.SpikeInjector(),
       label='spike_injector_forward')
```

Additional Parameters

As two or more programs cannot listen to the same UDP port for notification, the following additional parameters can be provided to the activate_live_output_for function or as parameters to the SpikeInjector.

database_notify_host	The host to which to send the notification that the database is ready to read. By default, the IP address of the machine executing the script is used. The same IP address can be used multiple times if the same host is running multiple senders or receivers or the same program is used for both sending and receiving.
database_notify_port_num	The UDP port number to which to send the notification that the database is ready to read. By default, port 19999 is used. This should be different for each sender or receiver; the same port can be used if the sender or receiver is the same as that used for other I/O.
notify	By default this is set to True, but if set to False, the above parameters are ignored and no notifications will be sent about the database for this sender or receiver. This can be useful to avoid sending messages to devices that can't be made to

	support the protocol.

Python Live Receiver / Injector

A SpynnakerLiveSpikesConnection is provided to perform the operation of sending or receiving spikes.

Receiver

The following block of code creates a live packet receiver to receive spikes from a live simulation:

```
1   # declare python code when received spikes for a timer tick
2   def receive_spikes(label, time, neuron_ids):
3       for neuron_id in neuron_ids:
4       print “Received spike at time {} from {}-{}”.format(
5           time, label, neuron_id)
6
7   # import python live spike connection
8   from spynnaker_external_devices_plugin.pyNN.connections.\
9                                 spynnaker_live_spikes_connection     import
SpynnakerLiveSpikesConnection
10
11  # set up python live spike connection
12  live_spikes_connection = SpynnakerLiveSpikesConnection(
13      receive_labels=[“receiver”])
14
15  # register python receiver with live spike connection
16  live_spikes_connection.add_receive_callback(“receiver”, receive_spikes)
```

1. Lines 1 to 5 creates a function that takes as its input all the neuron ids that fired at a specific time, from the population with the given label. From here, it generates a print message for each neuron.
2. Lines 7 to 9 imports the python support for live injection/live retrieval. The SpynnakerLiveSpikesConnection handles both live retrieval and live injection.
3. Lines 11 to 13 instantiates the SpynnakerLiveSpikesConnection, and informs the connection that it will receive data under the label “receiver”.
4. Lines 15 to 16 informs the connection that any packets being received with the “receiver” label need to be forwarded to the function receive_spikes defined on lines 1 to 5.

This script must be run in advance of the script that sets up the simulation. The SpynnakerLiveSpikesConnection will listen for the simulation script to complete the setup operations and so starts synchronized with the simulation. It is possible to run the reception of spikes within the same script as the simulation; to do this, ensure that the above code is placed before the call to run().

Sender

The following block of code creates a live packet injector:

```
1  # create python injector
2  def send_spike(label, sender):
3      sender.send_spike(label, 0, send_full_keys=True)
5
```

```
6  # import python injector connection
7  from spynnaker_external_devices_plugin.pyNN.connections.\
8  spynnaker_live_spikes_connection import SpynnakerLiveSpikesConnection
9
10 # set up python injector connection
11 live_spikes_connection = SpynnakerLiveSpikesConnection(
12     send_labels=["spike_sender"])
13
14 # register python injector with injector connection
15 live_spikes_connection.add_start_callback("spike_sender", send_spike)
```

1. Lines 1 to 3 create a function that will be called when the simulation starts, allowing the synchronized sending of spikes.
2. Lines 6 to 8 imports the python support for live injection/live retrieval. The SpynnakerLiveSpikesConnection handles both live retrieval and live injection.
3. Lines 10 to 12 instantiates the SpynnakerLiveSpikesConnection, and informs the connection it will inject data via the label spike_sender.
4. Lines 14 to 15 informs the connection that when the simulation starts, to call the send_spike function defined on lines 1 to 3.

As with the live reception script, this must be called before the simulation script, or before run() in the simulation script.

Multiple Senders and Receivers

If you need more than one SpynnakerLiveSpikesConnection on the same host, the connection can take an additional parameter specifying the local port to listen on for notifications from the simulation, by specifying the local_port parameter in the constructor e.g.:

```
live_spikes_connection_1 = SpynnakerLiveSpikesConnection(
    receive_labels=["receiver"], local_port=19996)
live_spikes_connection_2 = SpynnakerLiveSpikesConnection(
    send_labels=["sender"], local_port=19997)
```

Note that you must then also tell the simulation side that these ports are in use as described previously. This can be done when calling activate_live_output_for for the population or when creating a SpikeInjector by specifying the database_notify_port_num parameter e.g.

```
activate_live_output_for(receiver, database_notify_port_num=19996)
injector = p.Population(1, p.external_devices.SpikeInjector(
   database_notify_port_num=19997), label="sender")
```

Caveats

To use the live injection and retrieval functionality only supports the use of the Ethernet connection, which means that there is a limited bandwidth of a maximum of approx 30 MB/s. This bandwidth is shared between both types of functionality, as well as system support for certain types of neural models, such as the SpikeSourceArray.

Furthermore, this functionality depends upon the lossy communication fabric of the SpiNNaker machine. This means that even though a neuron fires a spike you may not see it via the live retrieval functionality. If you need to ensure you receive every packet that has been transmitted, we recommend using the standard PyNN functionality.

By using this functionality, you are making your script non portable between different simulators. The activate_live_output_for(<pop_object>) and SpikeInjector models are not supported by other PyNN backends (such as Nest, Brian etc).

Finally, this functionality uses a number of additional SpiNNaker cores. Therefore a network which would just fit onto your SpiNNaker machine before would likely fail to fit on the machine when these functionalities are added in.

Tasks

Task 1.1: A synfire chain with live output [Easy]

This task will create a synfire chain which displays live output. Start with the synfire chain from PyNNExamples.

1. Call activate_live_output_for(<pop_object>) on the synfire population.
2. Build a python receiver function that prints out the neuron ids for the population.
3. Import and instantiate a SpynnakerLiveSpikesConnection connection.
4. Link a receive callback to the python receiver function and print when a spike is received.

Task 1.2: A synfire chain with live input [Easy]

This task will create a synfire chain which is stimulated from a spike generated on the host and then injected into the simulation. Start with the synfire chain from PyNNExamples.

1. Remove the spike source array population.
2. Replace it with the SpikeInjector population.
3. Build a python injector function which sends a spike when called.
4. Import and instantiate a SpynnakerLiveSpikesConnection.
5. Link a start callback to the python injector function.

Task 1.3: A synfire chain with live input and output [Easy]

Take the code from the previous 2 tasks and integrate them together to produce one that injects and streams the packets back to the terminal.

1. Remember that you can use both the recieve_labels and send_labels of the same SpynnakerLiveSpikesConnection.

Task 2: 2 co-operative live synfire chains [Medium]

This task will create a 2 synfire chains which activate each other, but via python I/O.

1. Create a synfire chain activated by the first neuron of a SpikeInjector with 2 neurons.
2. Create a second synfire chain activated by the second neuron of the same SpikeInjector.
3. Activate live output for the two synfire chains.
4. Create a function that will run on start that will send a spike from the first neuron of the SpikeInjector.
5. Create a function to receive spikes from the first synfire chain; when a spike is received from the last neuron of this synfire chain, send a spike from the second neuron of the SpikeInjector.

6. Create a function to receive spikes from the second synfire chain; when a spike is received from the last neuron of this synfire chain, send a spike from the first neuron of the SpikeInjector.
7. Create a SpynnakerLiveSpikesConnection instance for the SpikeInjector and the two synfire chains.
8. Add each of the above functions as callbacks to the appropriate events.
9. Run the network and display the results.

Task 3: Synfire chain with multiple I/O scripts [Hard]

This task will work similarly to the previous tasks, but will split the parts into multiple scripts, which then need to be started in the correct order for it to work. This will be similar to having an external visualiser or external device which needs to be started before the simulation starts.

1. Create a script which uses the SpynnakerLiveSpikesConnection to receive spikes from a population specified on the command line, using a database port number specified on the command line and prints the spikes to the terminal (hint: use sys.argv[1] to get the first argument to a Python script).
2. Create a script which uses the SpynnakerLiveSpikeConnection to send spikes to a population specified on the command line, using a database port number specified on the command line, sending a spike with a neuron id specified on the command line after a random interval after the simulation starts.
3. Create a script which sets up a SpikeInjector connected to a synfire population, and which produces live output. This should set up two database notification ports each of which are specified on the command line, one for the SpikeInjector and one for the live output.
4. Run the scripts in the correct order specifying the correct parameters.