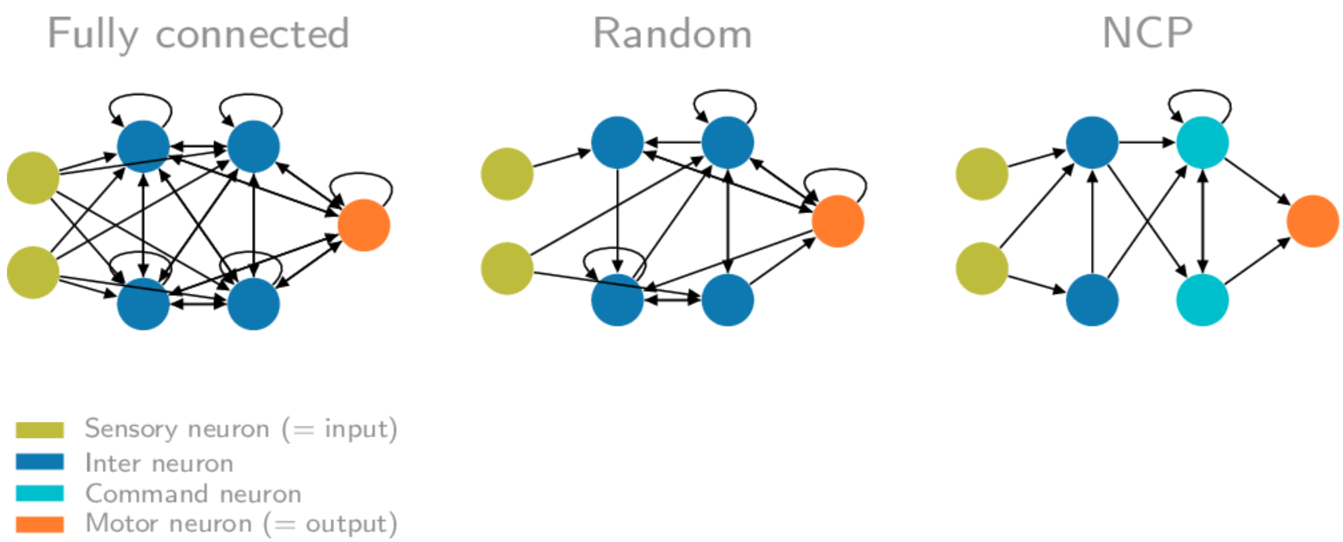
The basics of Neural Circuit Policies

In this tutorial we will build three recurrent neural networks based on the LTC model:

- A fully-connected network
- A sparse, randomly wired network
- A sparse, structured network based on the NCP principles



We will train these networks on some generated time-series and compare their training performance.

```
# Install dependencies if they are not installed yet
!pip install seaborn ncps
```

```
Looking in indexes: https://pypi.org/simple, https://us-python.pkg.dev/colab-wheels/public/simple/
Requirement already satisfied: seaborn in /usr/local/lib/python3.8/dist-packages (0.11.2)
Collecting ncps
  Downloading ncps-0.0.7-py3-none-any.whl (44 kB)
     ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 44.8/44.8 KB 1.2 MB/s eta 0:00:00
Requirement already satisfied: matplotlib>=2.2 in /usr/local/lib/python3.8/dist-packages (from seaborn) (3.2.2)
Requirement already satisfied: scipy>=1.0 in /usr/local/lib/python3.8/dist-packages (from seaborn) (1.7.3)
Requirement already satisfied: numpy>=1.15 in /usr/local/lib/python3.8/dist-packages (from seaborn) (1.21.6)
Requirement already satisfied: pandas>=0.23 in /usr/local/lib/python3.8/dist-packages (from seaborn) (1.3.5)
Requirement already satisfied: packaging in /usr/local/lib/python3.8/dist-packages (from ncps) (23.0)
Requirement already satisfied: future in /usr/local/lib/python3.8/dist-packages (from ncps) (0.16.0)
Requirement already satisfied: pyparsing!=2.0.4,!=2.1.2,!=2.1.6,>=2.0.1 in /usr/local/lib/python3.8/dist-packages (from mat
Requirement already satisfied: python-dateutil>=2.1 in /usr/local/lib/python3.8/dist-packages (from matplotlib>=2.2->seabor
Requirement already satisfied: cycler>=0.10 in /usr/local/lib/python3.8/dist-packages (from matplotlib>=2.2->seaborn) (0.11
Requirement already satisfied: kiwisolver>=1.0.1 in /usr/local/lib/python3.8/dist-packages (from matplotlib>=2.2->seaborn)
Requirement already satisfied: pytz>=2017.3 in /usr/local/lib/python3.8/dist-packages (from pandas>=0.23->seaborn) (2022.7.
Requirement already satisfied: six>=1.5 in /usr/local/lib/python3.8/dist-packages (from python-dateutil>=2.1->matplotlib>=2
Installing collected packages: ncps
Successfully installed ncps-0.0.7
```

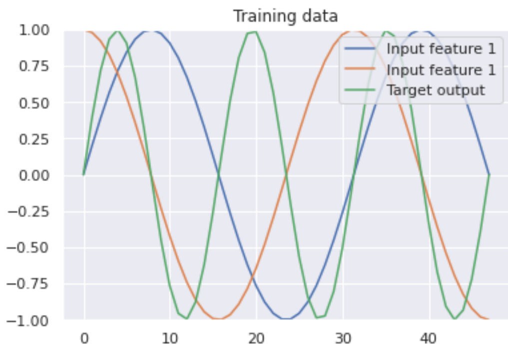
```
import numpy as np
import os
from tensorflow import keras
from ncps import wirings
from ncps.tf import LTC
import matplotlib.pyplot as plt
import seaborn as sns
```

Generating synthetic sinusoidal training data

```
N = 48 # Length of the time-series
# Input feature is a sine and a cosine wave
data_x = np.stack(
    [np.sin(np.linspace(0, 3 * np.pi, N)), np.cos(np.linspace(0, 3 * np.pi, N))], axis=1
)
```

```
data_x = np.expand_dims(data_x, axis=0).astype(np.float32)  # Add batch dimension
# Target output is a sine with double the frequency of the input signal
data_y = np.sin(np.linspace(0, 6 * np.pi, N)).reshape([1, N, 1]).astype(np.float32)
print("data_x.shape: ", str(data_x.shape))
print("data_y.shape: ", str(data_y.shape))

# Let's visualize the training data
sns.set()
plt.figure(figsize=(6, 4))
plt.plot(data_x[0, :, 0], label="Input feature 1")
plt.plot(data_x[0, :, 1], label="Input feature 1")
plt.plot(data_y[0, :, 0], label="Target output")
plt.ylim((-1, 1))
plt.title("Training data")
plt.legend(loc="upper right")
plt.show()
```

```
data_x.shape:  (1, 48, 2)
data_y.shape:  (1, 48, 1)
```



The LTC model

The `ncps` package is composed of two main parts:

- The LTC model as a `tf.keras.layers.Layer` RNN
- An wiring architecture for the LTC cell above

The wiring could be fully-connected (all-to-all) or sparsely designed using the NCP principles introduced in the paper.

Note that as the LTC model is expressed in the form of a system of ordinary differential equations in time, any instance of it is inherently a recurrent neural network (RNN). That's why this simple example considers a sinusoidal time-series.

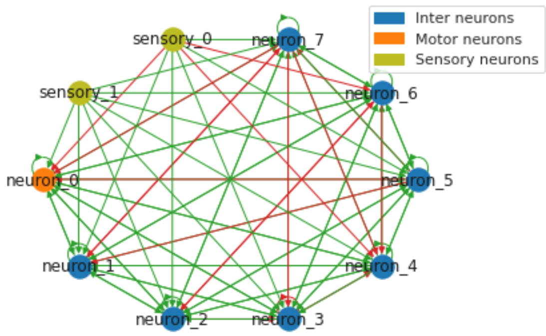
Our first LTC model with fully-connected wiring

```
fc_wiring = wirings.FullyConnected(8, 1)  # 8 units, 1 of which is a motor neuron

model = keras.models.Sequential(
    [
        keras.layers.InputLayer(input_shape=(None, 2)),
        LTC(fc_wiring, return_sequences=True),
    ]
)
model.compile(
    optimizer=keras.optimizers.Adam(0.01), loss='mean_squared_error'
)

model.summary()
```

```
Model: "sequential"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 ltc (LTC)                   (None, None, 1)           350

=================================================================
Total params: 350
Trainable params: 350
Non-trainable params: 0
_________________________________________________________________
```

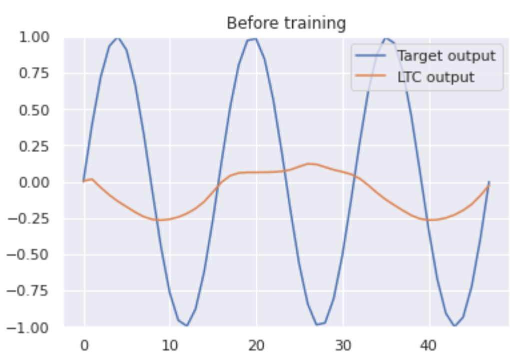
Draw the wiring diagram of the network

```
sns.set_style("white")
plt.figure(figsize=(6, 4))
legend_handles = fc_wiring.draw_graph(draw_labels=True)
plt.legend(handles=legend_handles, loc="upper center", bbox_to_anchor=(1, 1))
sns.despine(left=True, bottom=True)
plt.tight_layout()
plt.show()
```



Visualizing the prediction of the network before training

```
# Let's visualize how LTC initialy performs before the training
sns.set()
prediction = model(data_x).numpy()
plt.figure(figsize=(6, 4))
plt.plot(data_y[0, :, 0], label="Target output")
plt.plot(prediction[0, :, 0], label="LTC output")
plt.ylim((-1, 1))
plt.title("Before training")
plt.legend(loc="upper right")
plt.show()
```



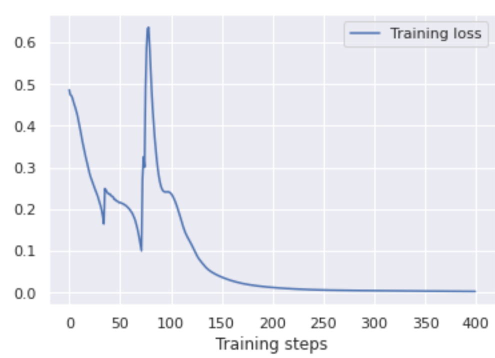
Training the model

```
# Train the model for 400 epochs (= training steps)
hist = model.fit(x=data_x, y=data_y, batch_size=1, epochs=400,verbose=1)
```

```
1/1 [==============================] - 0s 67ms/step - loss: 0.0029
Epoch 379/400
1/1 [==============================] - 0s 58ms/step - loss: 0.0029
Epoch 380/400
1/1 [==============================] - 0s 65ms/step - loss: 0.0029
Epoch 381/400
1/1 [==============================] - 0s 71ms/step - loss: 0.0028
Epoch 382/400
1/1 [==============================] - 0s 62ms/step - loss: 0.0028
Epoch 383/400
1/1 [==============================] - 0s 64ms/step - loss: 0.0028
Epoch 384/400
1/1 [==============================] - 0s 64ms/step - loss: 0.0028
Epoch 385/400
1/1 [==============================] - 0s 64ms/step - loss: 0.0028
Epoch 386/400
1/1 [==============================] - 0s 68ms/step - loss: 0.0028
Epoch 387/400
1/1 [==============================] - 0s 61ms/step - loss: 0.0028
Epoch 388/400
1/1 [==============================] - 0s 79ms/step - loss: 0.0027
Epoch 389/400
1/1 [==============================] - 0s 69ms/step - loss: 0.0027
Epoch 390/400
1/1 [==============================] - 0s 61ms/step - loss: 0.0027
Epoch 391/400
1/1 [==============================] - 0s 77ms/step - loss: 0.0027
Epoch 392/400
1/1 [==============================] - 0s 65ms/step - loss: 0.0027
Epoch 393/400
1/1 [==============================] - 0s 68ms/step - loss: 0.0027
Epoch 394/400
1/1 [==============================] - 0s 69ms/step - loss: 0.0026
Epoch 395/400
1/1 [==============================] - 0s 61ms/step - loss: 0.0026
Epoch 396/400
1/1 [==============================] - 0s 70ms/step - loss: 0.0026
Epoch 397/400
1/1 [==============================] - 0s 64ms/step - loss: 0.0026
Epoch 398/400
1/1 [==============================] - 0s 62ms/step - loss: 0.0026
Epoch 399/400
1/1 [==============================] - 0s 64ms/step - loss: 0.0026
Epoch 400/400
1/1 [==============================] - 0s 68ms/step - loss: 0.0026
```

Plotting the training loss

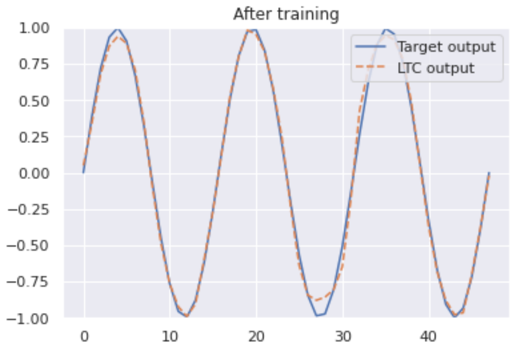
```
# Let's visualize the training loss
sns.set()
plt.figure(figsize=(6, 4))
plt.plot(hist.history["loss"], label="Training loss")
plt.legend(loc="upper right")
plt.xlabel("Training steps")
plt.show()
```



Plotting the prediction of the trained model

```
# How does the trained model now fit to the sinusoidal function?
prediction = model(data_x).numpy()
plt.figure(figsize=(6, 4))
plt.plot(data_y[0, :, 0], label="Target output")
```

```
plt.plot(prediction[0, :, 0], label="LTC output",linestyle="dashed")
plt.ylim((-1, 1))
plt.legend(loc="upper right")
plt.title("After training")
plt.show()
```



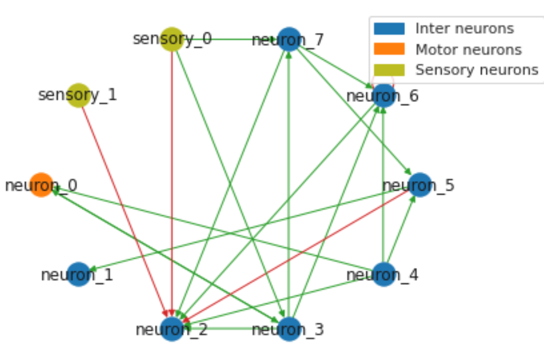
Other wiring architectures

Next, let's see how we can define other wiring architectures and compare them to the fully-connected network above

Random network with 75% sparsity

Let's create a randomly wired network where 75% of all synapses are removed.

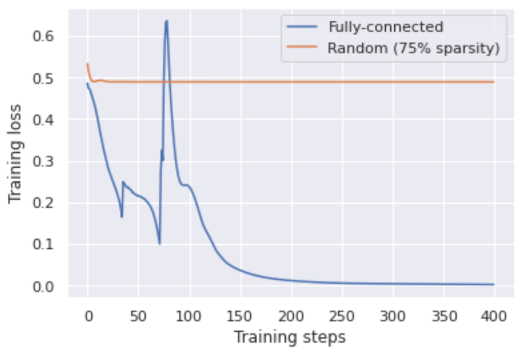
```
# Define LTC cell and wiring architecture
rnd_wiring = wirings.Random(8, 1,sparsity_level=0.75)  # 8 units, 1 motor neuron

# Define Keras model
sparse_model = keras.models.Sequential(
    [
        keras.layers.InputLayer(input_shape=(None, 2)),
        LTC(rnd_wiring, return_sequences=True),
    ]
)
sparse_model.compile(
    optimizer=keras.optimizers.Adam(0.01), loss='mean_squared_error'
)

# Plot the wiring
sns.set_style("white")
plt.figure(figsize=(6, 4))
legend_handles = rnd_wiring.draw_graph(draw_labels=True)
plt.legend(handles=legend_handles, loc="upper center", bbox_to_anchor=(1, 1))
sns.despine(left=True, bottom=True)
plt.tight_layout()
plt.show()
```



Comparing random sparse vs fully-connected networks

```
hist_rand = sparse_model.fit(x=data_x, y=data_y, batch_size=1, epochs=400,verbose=0)
# This may take a while (training the sprase LTC model)
sns.set()
```

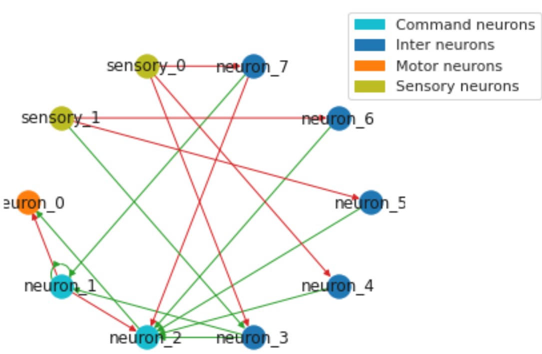
```
sns.set()
plt.figure(figsize=(6, 4))
plt.plot(hist.history["loss"], label="Fully-connected")
plt.plot(hist_rand.history["loss"], label="Random (75% sparsity)")
plt.legend(loc="upper right")
plt.xlabel("Training steps")
plt.ylabel("Training loss")
plt.show()
```



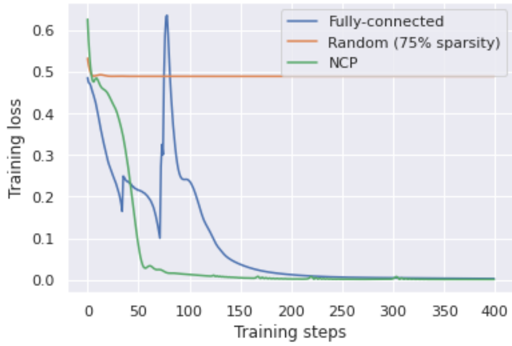
We see that the sparse model is not able to fit the sinusoidal signal as perfectly as the fully-connected architecture.

Neural Circuit Policy wiring architecture

```
ncp_arch = wirings.AutoNCP(8,1)

ncp_model = keras.models.Sequential(
    [
        keras.layers.InputLayer(input_shape=(None, 2)),
        LTC(ncp_arch, return_sequences=True),
    ]
)
ncp_model.compile(
    optimizer=keras.optimizers.Adam(0.01), loss='mean_squared_error'
)
sns.set_style("white")
plt.figure(figsize=(6, 4))
legend_handles = ncp_arch.draw_graph(draw_labels=True, neuron_colors={"command": "tab:cyan"})
plt.legend(handles=legend_handles, loc="upper center", bbox_to_anchor=(1.1, 1.1))
sns.despine(left=True, bottom=True)
plt.tight_layout()
plt.show()
```




```
hist_ncp = ncp_model.fit(x=data_x, y=data_y, batch_size=1, epochs=400,verbose=0)
# This may take a while (training the LTC model)
sns.set()
plt.figure(figsize=(6, 4))
plt.plot(hist.history["loss"], label="Fully-connected")
plt.plot(hist_rand.history["loss"], label="Random (75% sparsity)")
plt.plot(hist_ncp.history["loss"], label="NCP")
plt.legend(loc="upper right")
plt.xlabel("Training steps")
plt.ylabel("Training loss")
plt.show()
```



We see that the network with the NCP wiring architecture could fit the data as close as the fully-connected model.

Computing the sparsity of a NCP network

```
# Let's compare how many synapses the NCP network has compared to the fully-connected one
sparsity = 1 - ncp_arch.synapse_count /fc_wiring.synapse_count
print("Sparsity level is {:0.2f}%".format(100*sparsity))
```

```
    Sparsity level is 82.81%
```

The network with the NCP wiring performs as good as the fully-connected network but is even sparser than our random network tested above.