How to stack NCP with other types of layers

In this tutorial we will stack an NCP with a convolutional head, similar to the architecture for the end-to-end driving in the paper.

We make two minor simplifications though:

1. We won't train our model on the *driving-from-camera-images* dataset, but a 1D *collision avoidance from LIDAR signals* dataset intead. The reason is that the original dataset is quite large and takes a long time to train. Nonetheless, the taught concepts apply for image based data as well.
2. We won't seperate the feature-maps of the last convolutional layer as it was done in the paper. The reason is that it doesn't teach any NCP related concepts and might be a bit confusing.

```
# Install dependencies if they are not installed yet
!pip install -U 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 seaborn
  Downloading seaborn-0.12.2-py3-none-any.whl (293 kB)
     ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 293.3/293.3 KB 4.0 MB/s eta 0:00:00
Collecting ncps
  Downloading ncps-0.0.7-py3-none-any.whl (44 kB)
     ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 44.8/44.8 KB 2.9 MB/s eta 0:00:00
Requirement already satisfied: numpy!=1.24.0,>=1.17 in /usr/local/lib/python3.8/dist-packages (from seaborn) (1.21.6)
Requirement already satisfied: matplotlib!=3.6.1,>=3.1 in /usr/local/lib/python3.8/dist-packages (from seaborn) (3.2.2)
Requirement already satisfied: pandas>=0.25 in /usr/local/lib/python3.8/dist-packages (from seaborn) (1.3.5)
Requirement already satisfied: future in /usr/local/lib/python3.8/dist-packages (from ncps) (0.16.0)
Requirement already satisfied: packaging in /usr/local/lib/python3.8/dist-packages (from ncps) (23.0)
Requirement already satisfied: cycler>=0.10 in /usr/local/lib/python3.8/dist-packages (from matplotlib!=3.6.1,>=3.1->seabor
Requirement already satisfied: python-dateutil>=2.1 in /usr/local/lib/python3.8/dist-packages (from matplotlib!=3.6.1,>=3.1
Requirement already satisfied: kiwisolver>=1.0.1 in /usr/local/lib/python3.8/dist-packages (from matplotlib!=3.6.1,>=3.1->s
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: pytz>=2017.3 in /usr/local/lib/python3.8/dist-packages (from pandas>=0.25->seaborn) (2022.7.
Requirement already satisfied: six>=1.5 in /usr/local/lib/python3.8/dist-packages (from python-dateutil>=2.1->matplotlib!=3
Installing collected packages: ncps, seaborn
  Attempting uninstall: seaborn
    Found existing installation: seaborn 0.11.2
    Uninstalling seaborn-0.11.2:
      Successfully uninstalled seaborn-0.11.2
Successfully installed ncps-0.0.7 seaborn-0.12.2
```

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

Preparing the dataset

The dataset we are using considers the task of maneuvering a mobile robot to avoid obstacles in its path. Input data is obtained from a [Sick LMS 1xx laser rangefinder (LiDAR)]() mounted on the robot. Output variable is the steering direction as a variable in the range [-1,+1], i.e., -1 corresponding to turning left, 0 going straight, and +1 to turning right. Supervised training data was collected by manually steering the robot around the obstacles on 29 different tracks.

First, we will download the dataset.

```
from ncps.datasets import icra2020_lidar_collision_avoidance
# Download the dataset (already implemented in keras-ncp)
(x_train, y_train), (x_valid, y_valid) = icra2020_lidar_collision_avoidance.load_data()
print("x_train", str(x_train.shape))
print("y_train", str(y_train.shape))
```

```
Downloading file 'https://github.com/mlech26l/icra_lds/raw/master/icra2020_imitation_data_packed.npz'
x_train (678, 32, 541, 1)
y_train (678, 32, 1)
```

Note that there is no **test-set**. We are dealing here with a robotic control task were each action influences the future observations, which is very different from a *classify-and-forget* setting used in image classification tasks.

The environment feedback cannot be adequatly model during supervised training, so instead of evaluating the model on a test-set, we would need to **test the model live on the robot** to measure its performance.

Consequently, we will just monitor the metrics on the **validation-set** to give us some rough estimation of how well the model would perform in reality.

Anyways, let's plot a few samples of the training set to understand what problem we are dealing with here

```
def plot_lidar(lidar,ax):
  # Helper function for plotting polar-based lidar data
  angles = np.linspace(-2.35,2.35,len(lidar))
  x = lidar*np.cos(angles)
  y = lidar*np.sin(angles)
  ax.plot(y,x)
  ax.scatter([0],[0],marker="^",color="black")
  ax.set_xlim((-6,6))
  ax.set_ylim((-2,6))


sns.set()
fig,(ax1,ax2,ax3) = plt.subplots(1,3,figsize=(14,4))
plot_lidar(x_train[0,0,:,0],ax1)
plot_lidar(x_train[0,12,:,0],ax2)
plot_lidar(x_train[9,0,:,0],ax3)
ax1.set_title("Label: {:0.2f}".format(y_train[0,0,0]))
ax2.set_title("Label: {:0.2f}".format(y_train[0,12,0]))
ax3.set_title("Label: {:0.2f}".format(y_train[9,0,0]))
fig.suptitle("LIDAR collision avoidance training examples")
fig.show()
```

Bulding a stacked-NCP model

Here, we will create a neural network consisting of a feed-forward followed by a recurrent sub-model:

wirings.png

The input data are provided as a time-series where at each time-step we observe a full laser rangefinder scan. The network then feeds the LIDAR scan through the feedforward part to obtain a 32-dimensional latent representation of the current input. The recurrent NCP then takes this latant feature as input and updates its internal state and output prediction.

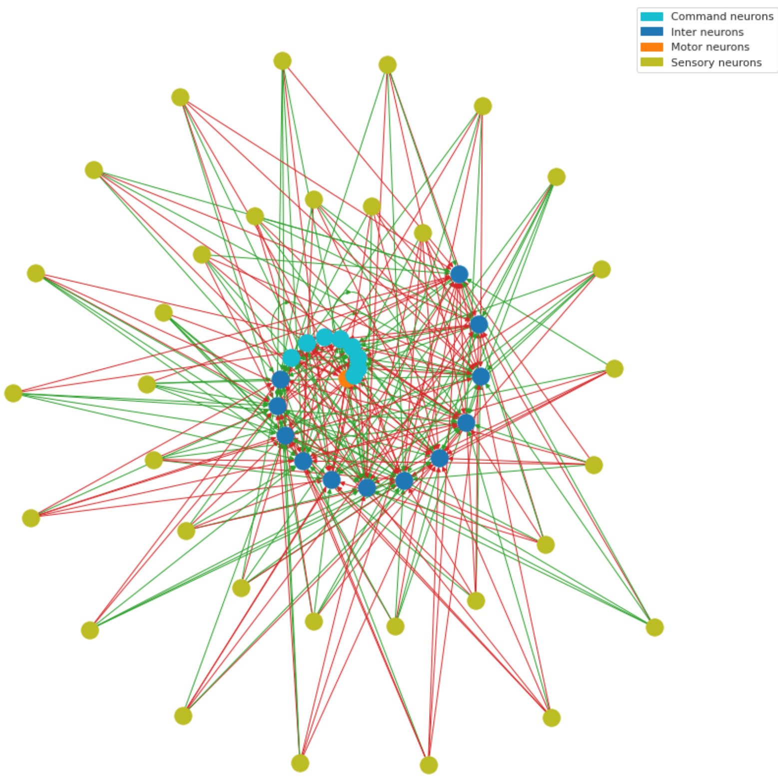
```
N = x_train.shape[2]
channels = x_train.shape[3]

wiring = AutoNCP(21,1)

# We need to use the TimeDistributed layer to independently apply the
# Conv1D/MaxPool1D/Dense over each time-step of the input time-series.
model = keras.models.Sequential(
    [
        keras.layers.InputLayer(input_shape=(None, N, channels)),
        keras.layers.TimeDistributed(
            keras.layers.Conv1D(18, 5, strides=3, activation="relu")
        ),
        keras.layers.TimeDistributed(
            keras.layers.Conv1D(20, 5, strides=2, activation="relu")
        ),
        keras.layers.TimeDistributed(keras.layers.MaxPool1D()),
        keras.layers.TimeDistributed(
            keras.layers.Conv1D(22, 5, activation="relu")
        ),
        keras.layers.TimeDistributed(keras.layers.MaxPool1D()),
        keras.layers.TimeDistributed(
            keras.layers.Conv1D(24, 5, activation="relu")
        ),
        keras.layers.TimeDistributed(keras.layers.Flatten()),
        keras.layers.TimeDistributed(keras.layers.Dense(32, activation="relu")),
        LTC(wiring, return_sequences=True),
    ]
)
model.compile(
    optimizer=keras.optimizers.Adam(0.01), loss="mean_squared_error",
)

model.summary(line_length=100)
```

```
Model: "sequential"
____________________________________________________________________________________________________
 Layer (type)                                Output Shape                            Param #
====================================================================================================
 time_distributed (TimeDistributed)          (None, None, 179, 18)                   108

 time_distributed_1 (TimeDistributed)        (None, None, 88, 20)                    1820

 time_distributed_2 (TimeDistributed)        (None, None, 44, 20)                    0

 time_distributed_3 (TimeDistributed)        (None, None, 40, 22)                    2222

 time_distributed_4 (TimeDistributed)        (None, None, 20, 22)                    0

 time_distributed_5 (TimeDistributed)        (None, None, 16, 24)                    2664

 time_distributed_6 (TimeDistributed)        (None, None, 384)                       0

 time_distributed_7 (TimeDistributed)        (None, None, 32)                        12320

 ltc (LTC)                                   (None, None, 1)                         4581

====================================================================================================
Total params: 23,715
```

```
Trainable params: 23,715
Non-trainable params: 0
_________________________________________________________________
```

Let's draw the NCP wiring of our model

```
sns.set_style("white")
plt.figure(figsize=(12, 12))
legend_handles = wiring.draw_graph(layout='spiral',neuron_colors={"command": "tab:cyan"})
plt.legend(handles=legend_handles, loc="upper center", bbox_to_anchor=(1, 1))
sns.despine(left=True, bottom=True)
plt.tight_layout()
plt.show()
```



Training the stacked-NCP model

Before training the model, we first evaluate how well it performs on the validation set

```
model.evaluate(x_valid, y_valid)
```

```
8/8 [==============================] - 3s 101ms/step - loss: 0.2345
0.2344837635755539
```

```
model.fit(
    x=x_train, y=y_train, batch_size=32, epochs=20, validation_data=(x_valid, y_valid)
)
```

```
Epoch 1/20
22/22 [==============================] - 24s 432ms/step - loss: 0.2107 - val_loss: 0.1748
Epoch 2/20
22/22 [==============================] - 9s 399ms/step - loss: 0.2039 - val_loss: 0.1770
Epoch 3/20
22/22 [==============================] - 10s 455ms/step - loss: 0.2037 - val_loss: 0.1752
Epoch 4/20
22/22 [==============================] - 9s 432ms/step - loss: 0.2032 - val_loss: 0.1751
Epoch 5/20
22/22 [==============================] - 10s 445ms/step - loss: 0.2019 - val_loss: 0.1742
Epoch 6/20
22/22 [==============================] - 8s 363ms/step - loss: 0.1998 - val_loss: 0.1647
Epoch 7/20
22/22 [==============================] - 9s 431ms/step - loss: 0.1964 - val_loss: 0.1624
Epoch 8/20
22/22 [==============================] - 9s 419ms/step - loss: 0.1906 - val_loss: 0.1918
Epoch 9/20
22/22 [==============================] - 8s 372ms/step - loss: 0.2103 - val_loss: 0.1777
Epoch 10/20
22/22 [==============================] - 10s 460ms/step - loss: 0.2044 - val_loss: 0.1752
Epoch 11/20
22/22 [==============================] - 9s 433ms/step - loss: 0.2028 - val_loss: 0.1754
Epoch 12/20
22/22 [==============================] - 8s 379ms/step - loss: 0.2036 - val_loss: 0.1755
Epoch 13/20
22/22 [==============================] - 9s 401ms/step - loss: 0.2031 - val_loss: 0.1753
Epoch 14/20
22/22 [==============================] - 9s 423ms/step - loss: 0.2030 - val_loss: 0.1752
Epoch 15/20
22/22 [==============================] - 9s 406ms/step - loss: 0.2029 - val_loss: 0.1752
Epoch 16/20
22/22 [==============================] - 9s 412ms/step - loss: 0.2027 - val_loss: 0.1753
Epoch 17/20
22/22 [==============================] - 9s 426ms/step - loss: 0.2027 - val_loss: 0.1752
Epoch 18/20
22/22 [==============================] - 10s 440ms/step - loss: 0.2034 - val_loss: 0.1753
Epoch 19/20
22/22 [==============================] - 9s 403ms/step - loss: 0.2027 - val_loss: 0.1752
Epoch 20/20
22/22 [==============================] - 9s 422ms/step - loss: 0.2031 - val_loss: 0.1756
<keras.callbacks.History at 0x7f3580d666a0>
```

Now let's evaluate the performance of our model on the validation set after the training

```
model.evaluate(x_valid, y_valid)
```

```
8/8 [==============================] - 2s 203ms/step - loss: 0.1756
0.1755889654159546
```

✓ 2s completed at 3:49 PM