Neural Circuit Policies

Release 0.0.1

Mathias Lechner

Jul 28, 2023

CONTENTS

Neural Circuit Policies (NCPs) are machine learning models inspired by the nervous system of the nematode *C. elegans*. This package provides easy-to-use implementations of NCPs for PyTorch and Tensorflow.

```
pip3 install -U ncps
```

Example Pytorch example:

```
from ncps.torch import CfC

# a fully connected CfC network
rnn = CfC(input_size=20, units=50)
x = torch.randn(2, 3, 20) # (batch, time, features)
h0 = torch.zeros(2,50) # (batch, units)
output, hn = rnn(x,h0)
```

A Tensorflow example

```
# Tensorflow example
from ncps.tf import LTC
from ncps.wirings import AutoNCP

wiring = AutoNCP(28, 4) # 28 neurons, 4 outputs
model = tf.keras.models.Sequential(
    [
        tf.keras.layers.InputLayer(input_shape=(None, 2)),
        # LTC model with NCP sparse wiring
        LTC(wiring, return_sequences=True),
    ]
)
```

CHAPTER

ONE

USER'S GUIDE

1.1 Quickstart

Neural Circuit Policies are recurrent neural network models inspired by the nervous system of the nematode C. elegans. Compared to standard ML models, NCPs have

1. Neurons that are modeled by an ordinary differential equation
2. A sparse structured wiring

1.1.1 Neuron Models

The package currently provides two neuron models: LTC and CfC: The liquid time-constant (LTC) model is based on neurons in the form of differential equations interconnected via sigmoidal synapses. The term liquid time-constant comes from the property of LTCs that their timing behavior is adaptive to the input (how fast or slow they respond to some stimulus can depend on the specific input). Because LTCs are ordinary differential equations, their behavior can only be described over time. LTCs are universal approximators and implement causal dynamical models. However, the LTC model has one major disadvantage: to compute their output, we need a numerical differential equation-solver which seriously slows down their training and inference time. Closed-form continuous-time (CfC) models resolve this bottleneck by approximating the closed-form solution of the differential equation.

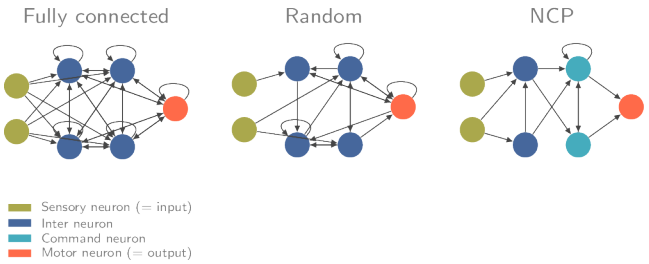
Note: Both the LTC and the CfC models are **recurrent neural networks** and possess a temporal state. Therefore, these models are applicable only to sequential or time-series data.

1.1.2 Wirings

We can use both models described above with a fully-connected wiring diagram by simply passing the number of neurons, i.e., as it is done in standard ML models such as LSTMs, GRU, MLPs, and Transformers.

```
from ncps.torch import CfC

# a fully connected CfC network
rnn = CfC(input_size=20, units=50)
```

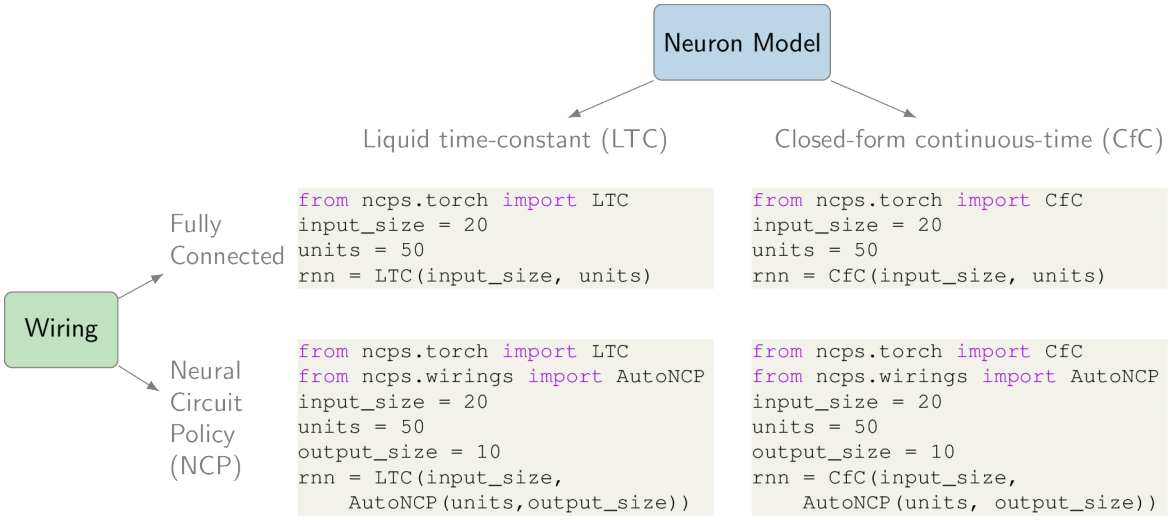
We can also specify sparse structured wirings in the form of a `ncps.wirings.Wiring` object. The Neural Circuit Policy (NCP) is the most interesting wiring paradigm provided in this package and comprises of a 4-layer recurrent connection principle of sensory, inter, command, and motor neurons.



The easiest way to create a NCP wiring is via the `AutoNCP` class, which requires defining the total number of neurons and the number of motor neurons (= output size).

```
from ncps.torch import CfC
from ncps.wirings import AutoNCP

wiring = AutoNCP(28, 4) # 28 neurons, 4 outputs
input_size = 20
rnn = CfC(input_size, wiring)
```

1.1.3 Diagram



1.2 Examples

1.2.1 First steps (Pytorch)

In this tutorial we will build small NCP model based on the LTC neuron model and train it on some synthetic sinusoidal data.

```
pip install seaborn ncps torch pytorch-lightning
```

```
import numpy as np
import torch.nn as nn
```

(continues on next page)

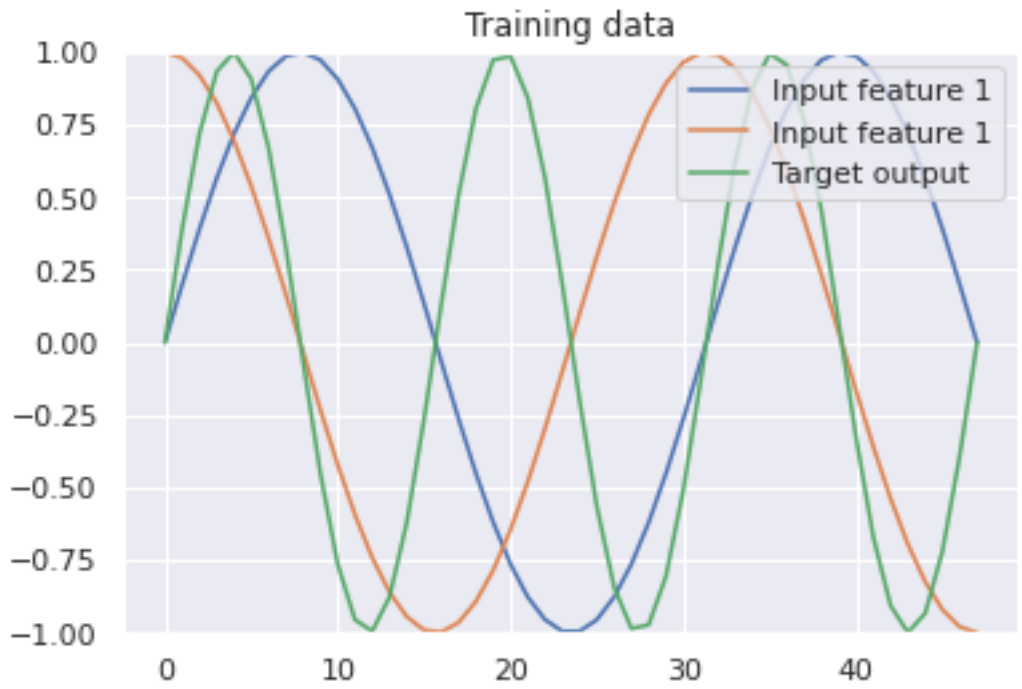
(continued from previous page)

```
from ncps.wirings import AutoNCP
from ncps.torch import LTC
import pytorch_lightning as pl
import torch
import torch.utils.data as data
```

Generating synthetic sinusoidal training data

```
import matplotlib.pyplot as plt
import seaborn as sns
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))
data_x = torch.Tensor(data_x)
data_y = torch.Tensor(data_y)
dataloader = data.DataLoader(
    data.TensorDataset(data_x, data_y), batch_size=1, shuffle=True, num_workers=4
)

# 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)
```



Pytorch-Lightning RNN training module

For training the model, we will use the pytorch-lightning high-level API. For that reason, we have to define a sequence learning module:

```
# LightningModule for training a RNNSequence module
class SequenceLearner(pl.LightningModule):
    def __init__(self, model, lr=0.005):
        super().__init__()
        self.model = model
        self.lr = lr

    def training_step(self, batch, batch_idx):
        x, y = batch
        y_hat, _ = self.model.forward(x)
        y_hat = y_hat.view_as(y)
        loss = nn.MSELoss()(y_hat, y)
        self.log("train_loss", loss, prog_bar=True)
        return {"loss": loss}

    def validation_step(self, batch, batch_idx):
        x, y = batch
        y_hat, _ = self.model.forward(x)
        y_hat = y_hat.view_as(y)
        loss = nn.MSELoss()(y_hat, y)

        self.log("val_loss", loss, prog_bar=True)
        return loss
```

(continues on next page)

(continued from previous page)

```
    def test_step(self, batch, batch_idx):
        # Here we just reuse the validation_step for testing
        return self.validation_step(batch, batch_idx)

    def configure_optimizers(self):
        return torch.optim.Adam(self.model.parameters(), lr=self.lr)
```

The LTC model with NCP wiring

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

- The LTC model as a `nn.module` object
- An wiring architecture for the LTC cell above

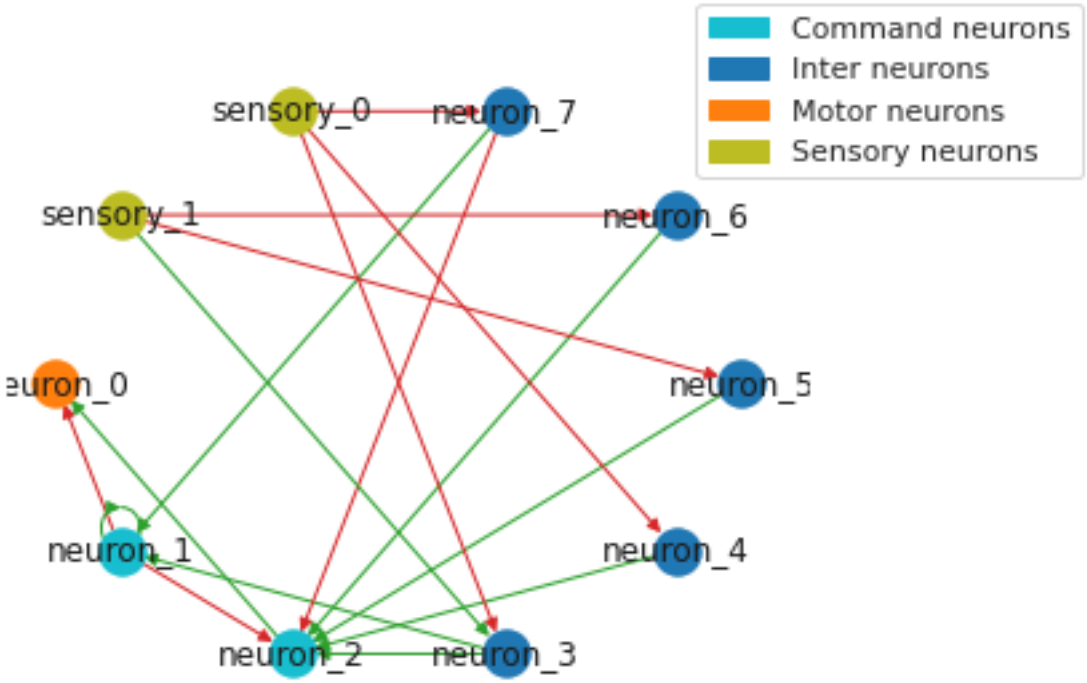
For the wiring we will use the `AutoNCP class, which creates a NCP wiring diagram by providing the total number of neurons and the number of outputs (16 and 1 in our case).

Note: Note that as the LTC model is expressed in the form of a system of [ordinary differential equations in time](https://arxiv.org/abs/2006.04439), any instance of it is inherently a recurrent neural network (RNN). That's why this simple example considers a sinusoidal time-series.

```
out_features = 1
in_features = 2

wiring = AutoNCP(16, out_features)  # 16 units, 1 motor neuron

ltc_model = LTC(in_features, wiring, batch_first=True)
learn = SequenceLearner(ltc_model, lr=0.01)
trainer = pl.Trainer(
    logger=pl.loggers.CSVLogger("log"),
    max_epochs=400,
    gradient_clip_val=1,  # Clip gradient to stabilize training
)
```

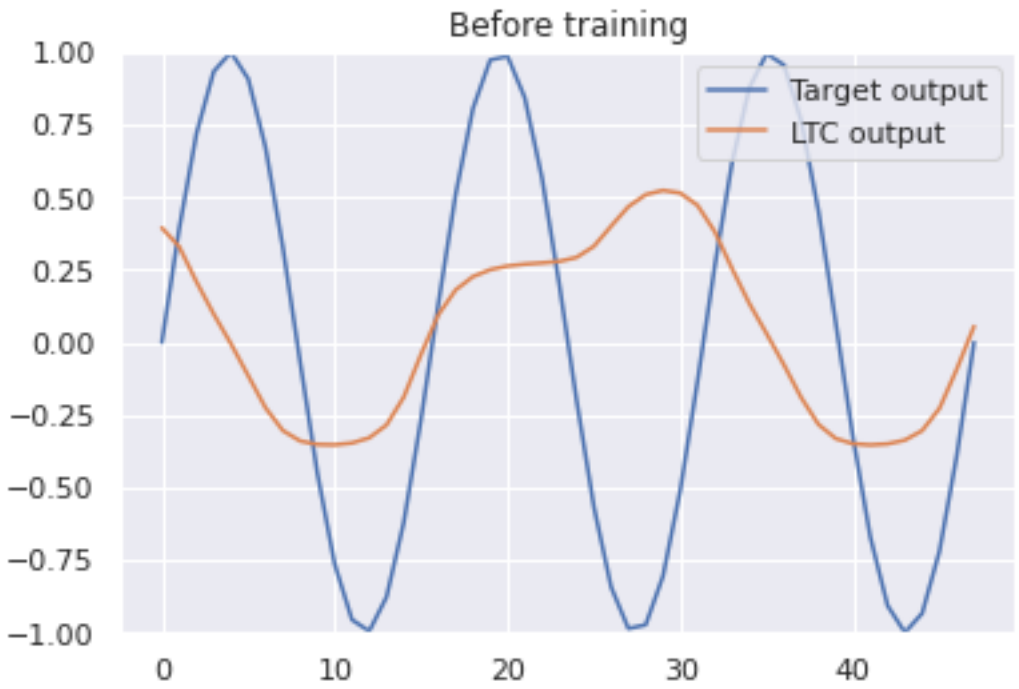
Draw the wiring diagram of the network

```
sns.set_style("white")
plt.figure(figsize=(6, 4))
legend_handles = wiring.draw_graph(draw_labels=True, 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()
```



Visualizing the prediction of the network before training

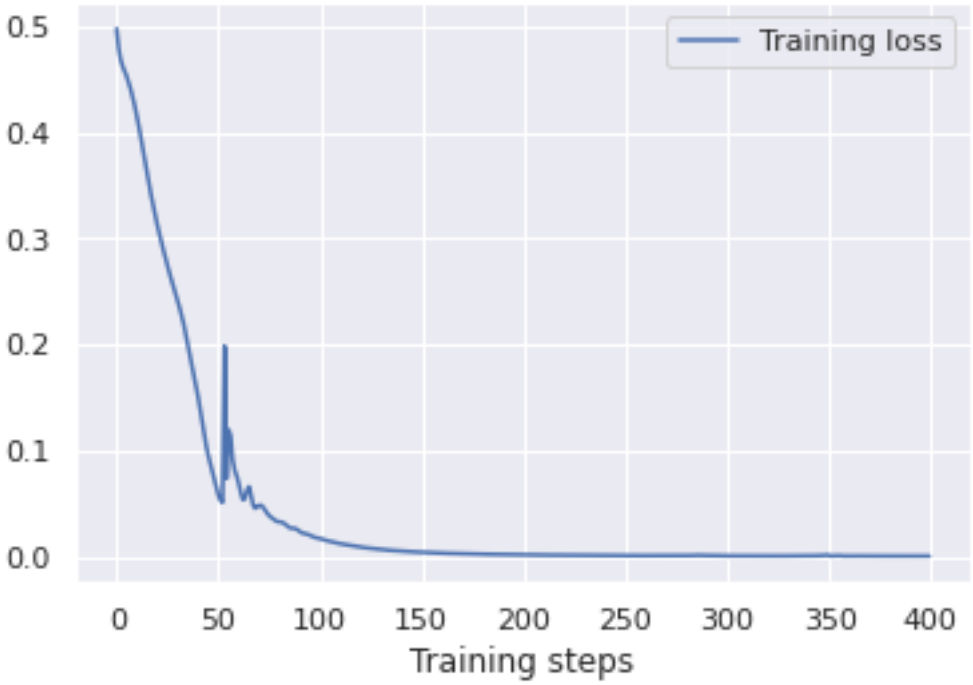
```
# Let's visualize how LTC initialy performs before the training
sns.set()
with torch.no_grad():
    prediction = ltc_model(data_x)[0].numpy()
plt.figure(figsize=(6, 4))
plt.plot(data_y[0, :, 0], label="Target output")
plt.plot(prediction[0, :, 0], label="NCP output")
plt.ylim((-1, 1))
plt.title("Before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



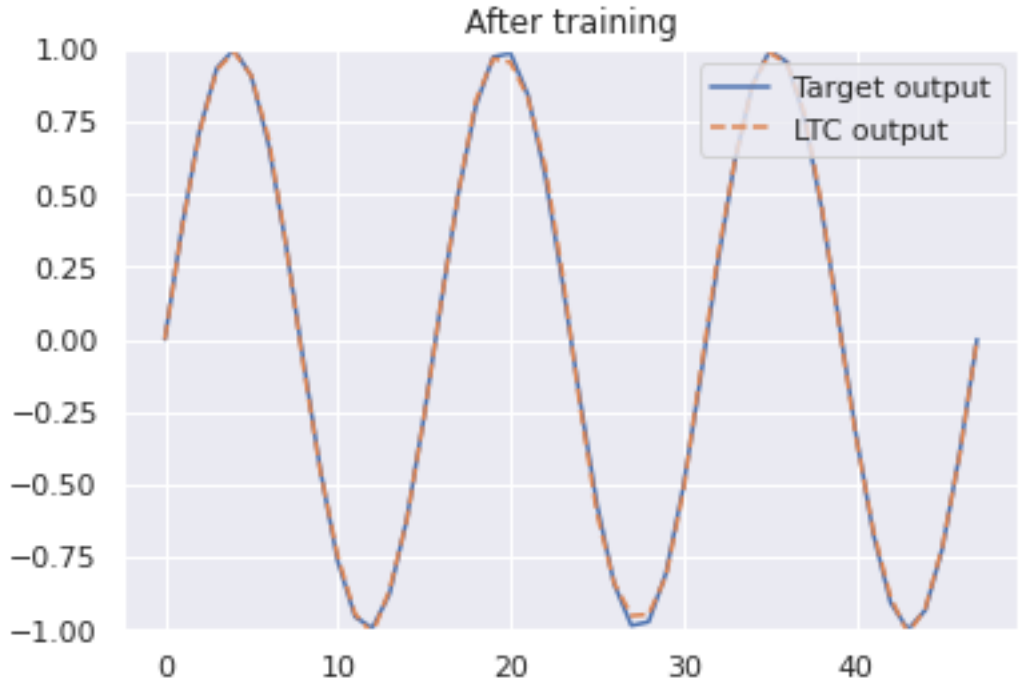
Training the model

```
# Train the model for 400 epochs (= training steps)
trainer.fit(learn, dataloader)
```

```
.... 1/1 [00:00<00:00, 4.91it/s, loss=0.000459, v_num=0, train_loss=0.000397]
```



```
# How does the trained model now fit to the sinusoidal function?
sns.set()
with torch.no_grad():
    prediction = ltc_model(data_x)[0].numpy()
plt.figure(figsize=(6, 4))
plt.plot(data_y[0, :, 0], label="Target output")
plt.plot(prediction[0, :, 0], label="NCP output")
plt.ylim((-1, 1))
plt.title("After training")
plt.legend(loc="upper right")
plt.show()
```



1.2.2 First steps (Tensorflow)

In this tutorial we will build small NCP model based on the LTC neuron model and train it on some synthetic sinusoidal data.

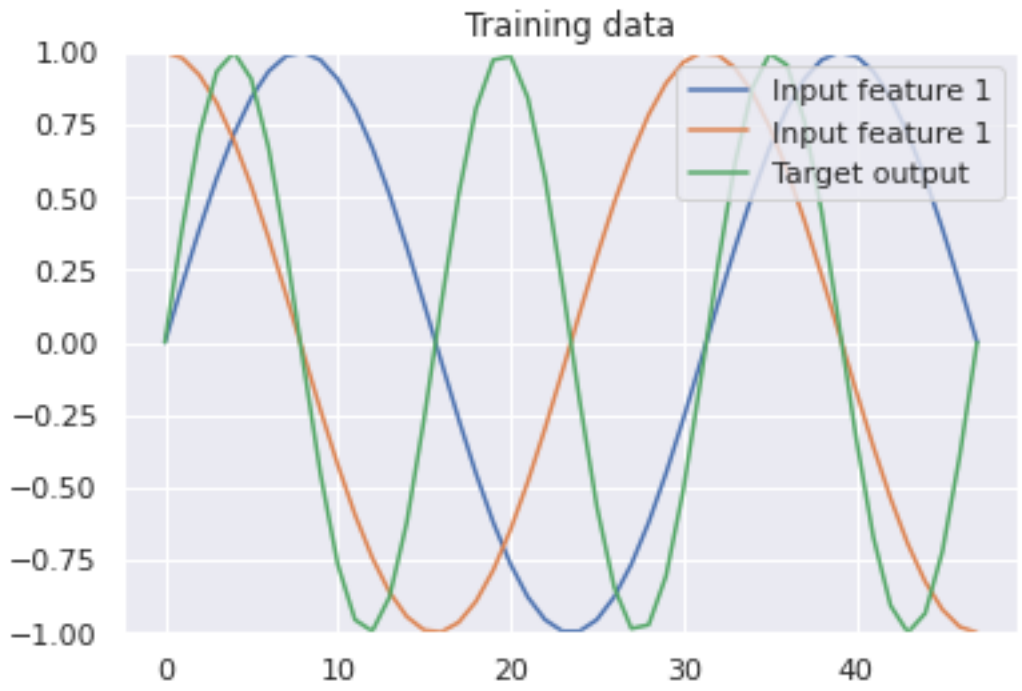
```
pip install seaborn ncps
```

```
import numpy as np
import os
from tensorflow import keras
from ncps import wirings
from ncps.tf import LTC
```

Generating synthetic sinusoidal training data

```
import matplotlib.pyplot as plt
import seaborn as sns

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 with NCP wiring

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

For the wiring we will use the `AutoNCP` class, which creates a NCP wiring diagram by providing the total number of neurons and the number of outputs (8 and 1 in our case).

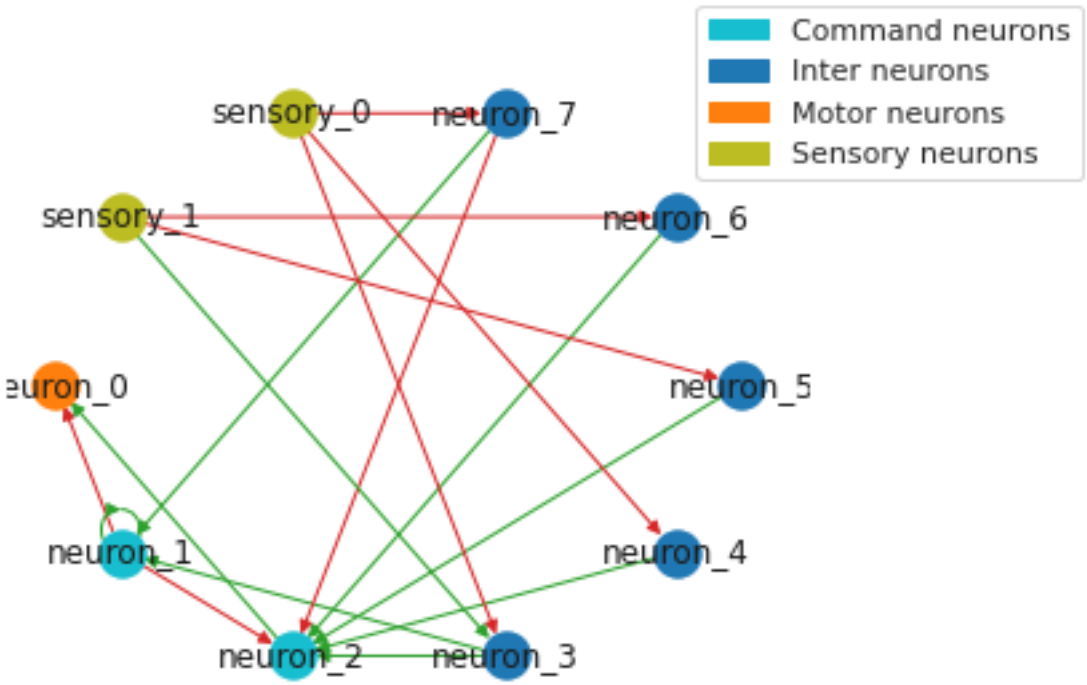
Note: Note that as the LTC model is expressed in the form of a system of [ordinary differential equations in time](https://arxiv.org/abs/2006.04439), any instance of it is inherently a recurrent neural network (RNN). That's why this simple example considers a sinusoidal time-series.

```
wiring = wirings.AutoNCP(8,1) # 8 neurons in total, 1 output (motor neuron)
model = keras.models.Sequential(
    [
        keras.layers.InputLayer(input_shape=(None, 2)),
        # here we could potentially add layers before and after the LTC network
        LTC(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
_________________________________________________________________
```

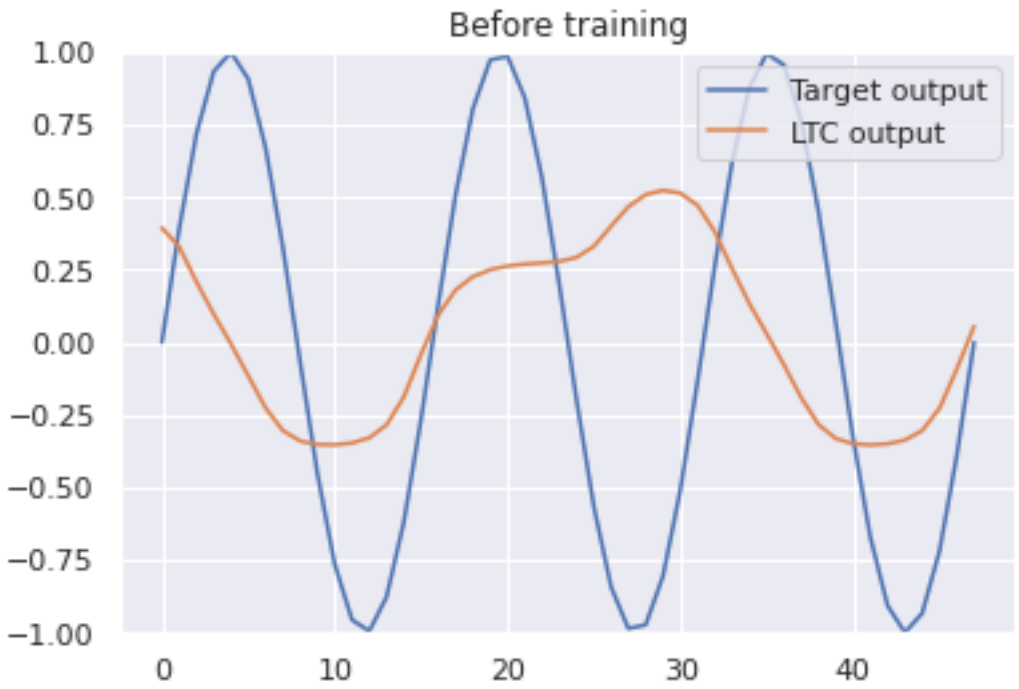
Draw the wiring diagram of the network

```
sns.set_style("white")
plt.figure(figsize=(6, 4))
legend_handles = wiring.draw_graph(draw_labels=True, 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()
```



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="NCP output")
plt.ylim((-1, 1))
plt.title("Before training")
plt.legend(loc="upper right")
plt.show()
```



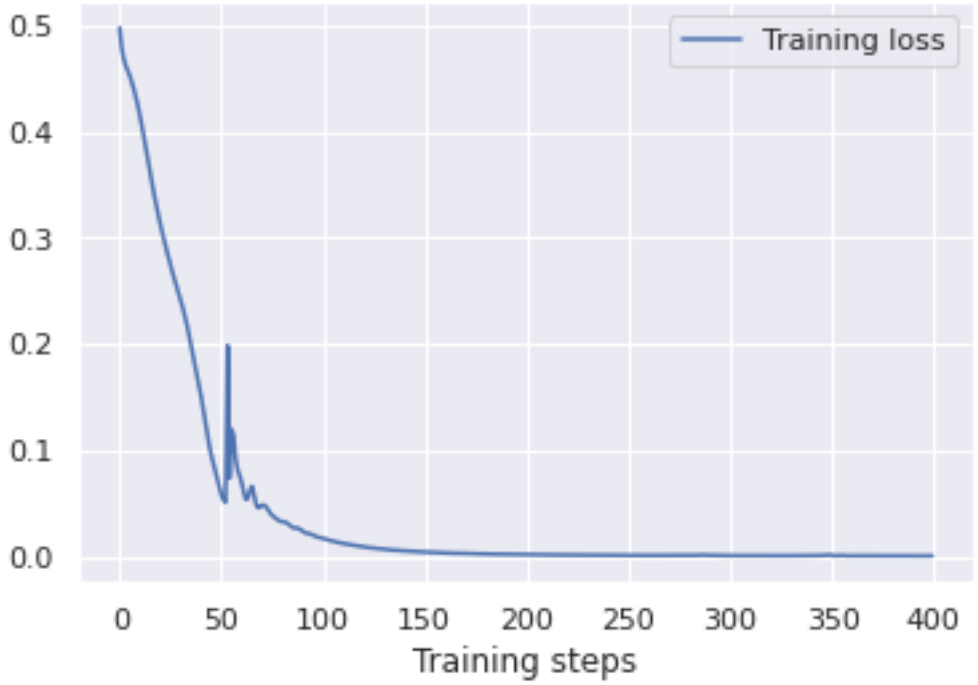
Training the model

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

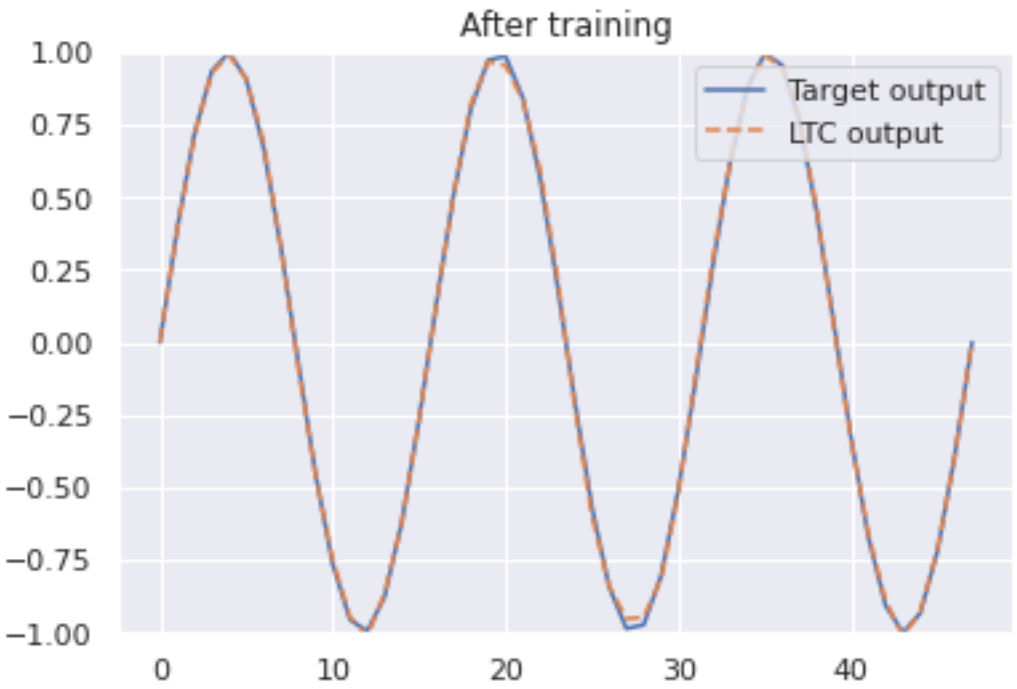
```
Epoch 1/400
1/1 [==============================] - 6s 6s/step - loss: 0.4980
Epoch 2/400
1/1 [==============================] - 0s 55ms/step - loss: 0.4797
Epoch 3/400
1/1 [==============================] - 0s 54ms/step - loss: 0.4686
Epoch 4/400
1/1 [==============================] - 0s 57ms/step - loss: 0.4623
Epoch 5/400
....
Epoch 395/400
1/1 [==============================] - 0s 63ms/step - loss: 2.3493e-04
Epoch 396/400
1/1 [==============================] - 0s 57ms/step - loss: 2.3593e-04
Epoch 397/400
1/1 [==============================] - 0s 64ms/step - loss: 2.3607e-04
Epoch 398/400
1/1 [==============================] - 0s 69ms/step - loss: 2.3487e-04
Epoch 399/400
1/1 [==============================] - 0s 73ms/step - loss: 2.3288e-04
Epoch 400/400
1/1 [==============================] - 0s 65ms/step - loss: 2.3024e-04
```

Plotting the training loss and the prediction of the model after training

```
# 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()
```



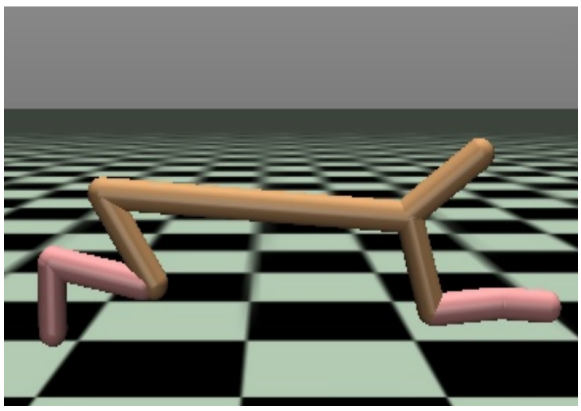

```
# How does the trained model now fit to the sinusoidal function?
prediction = model(data_x).numpy()
plt.figure(figsize=(6, 4))
plt.plot(data_y[0, :, 0], label="Target output")
plt.plot(prediction[0, :, 0], label="LTC output",linestyle="dashed")
plt.ylim((-1, 1))
plt.legend(loc="upper right")
plt.title("After training")
plt.show()
```



1.2.3 Partially Observabily Reinforcement Learning

In this guide, we will train a CfC network to solve a partially observable Markov decision process (POMDP). In particular, we consider a partially observable version of the HalfCheetah Mujoco environment. Moreover, we will also evalute the trained policy when the observation is corrupted by noise to test the robustness of the learned policy. This tutorial serves as a beginner's guide to using CfC networks in reinforcement learning, showing how to define a custom CfC network and how to use it with rllib, as well as demonstrating the the advantages of RNNs in partially observable environments.

Code is provided for TensorFlow and relies on ray[rllib] for it's proximal policy optimization (PPO) implementation and gymnasium[mujoco] for the HalfCheetah environment.



Setup and Requirements

Before we start, we need to install some packages

```
pip3 install ncps tensorflow "ray[rllib]" "gymnasium[mujoco]"
```

Partially Observable HalfCheetah

First, we need to create a partially observable version of the HalfCheetah environment. We do this by wrapping the original environment in a `gymnasium.ObservationWrapper`

```
import gymnasium
from gymnasium import spaces

import ray
from ray.tune.registry import register_env
from ray.rllib.models import ModelCatalog
from ray.rllib.algorithms.ppo import PPO
import time
import numpy as np

from ray.rllib.models.modelv2 import ModelV2
from ray.rllib.models.tf.recurrent_net import RecurrentNetwork
from ray.rllib.utils.annotations import override

import tensorflow as tf
import ncps.tf

import matplotlib.pyplot as plt


class PartialObservation(gymnasium.ObservationWrapper):
    def __init__(self, env: gymnasium.Env, obs_indices: list):
        gymnasium.ObservationWrapper.__init__(self, env)

        obsspace = env.observation_space
        self.obs_indices = obs_indices
        self.observation_space = spaces.Box(
            low=np.array([obsspace.low[i] for i in obs_indices]),
            high=np.array([obsspace.high[i] for i in obs_indices]),
            dtype=np.float32,
        )

        self._env = env

    def observation(self, observation):
        filter_observation = self._filter_observation(observation)
        return filter_observation

    def _filter_observation(self, observation):
        observation = np.array([observation[i] for i in self.obs_indices])
        return observation
```

Concretely, we will remove all joint velcities from the observation space.

```
def make_partial_observation_cheetah():
    return PartialObservation(
        gymnasium.make("HalfCheetah-v4"), [0, 1, 2, 3, 8, 9, 10, 11, 12]
    )
```

CfC Policy Network

Next, we will subclass `ray.rllib.models.tf.recurrent_net.RecurrentNetwork` to define our CfC policy network. The network comprises of two dense layers followed by a CfC layer.

```
class CustomRNN(RecurrentNetwork):
    """Example of using the Keras functional API to define a RNN model."""

    def __init__(
        self,
        obs_space,
        action_space,
        num_outputs,
        model_config,
        name,
        cell_size=64,
    ):
        super(CustomRNN, self).__init__(
            obs_space, action_space, num_outputs, model_config, name
        )
        self.cell_size = cell_size

        # Define input layers
        input_layer = tf.keras.layers.Input(
            shape=(None, obs_space.shape[0]),
            name="inputs",
        )
        state_in_h = tf.keras.layers.Input(shape=(cell_size,), name="h")
        seq_in = tf.keras.layers.Input(shape=(), name="seq_in", dtype=tf.int32)

        # Preprocess observation with a hidden layer and send to RNN
        self.preprocess_layers = tf.keras.models.Sequential(
            [
                tf.keras.Input((obs_space.shape[0],)),  # batch dimension is implicit
                tf.keras.layers.Dense(256, activation="silu"),
                tf.keras.layers.Dense(256, activation="silu"),
            ]
        )
        self.td_preprocess = tf.keras.layers.TimeDistributed(self.preprocess_layers)

        dense1 = self.td_preprocess(input_layer)
        rnn_out, state_h = ncps.tf.CfC(
            cell_size, return_sequences=True, return_state=True, name="rnn"
        )(
            inputs=dense1,
```

(continues on next page)

(continued from previous page)

```
            mask=tf.sequence_mask(seq_in),
            initial_state=[state_in_h],
        )
        logits = tf.keras.layers.Dense(
            self.num_outputs, activation=None, name="logits"
        )(rnn_out)
        values = tf.keras.layers.Dense(1, activation=None, name="values")(rnn_out)

        # Create the RNN model
        self.rnn_model = tf.keras.Model(
            inputs=[input_layer, seq_in, state_in_h],
            outputs=[logits, values, state_h],
        )
        self.rnn_model.summary()

    @override(RecurrentNetwork)
    def forward_rnn(self, inputs, state, seq_lens):
        model_out, self._value_out, h = self.rnn_model([inputs, seq_lens] + state)
        return model_out, [h]

    @override(ModelV2)
    def get_initial_state(self):
        return [
            np.zeros(self.cell_size, np.float32),
        ]

    @override(ModelV2)
    def value_function(self):
        return tf.reshape(self._value_out, [-1])
```

Evaluate Function

Next, we will define a function that evaluates the performance of a policy network with optional noise injected to the observations.

Note: Depending on the rllib version and installation the `apply_filter` may be already included in the `compute_single_action` function.

```
def run_closed_loop(
    algo, rnn_cell_size, n_episodes=10, pertubation_level=0.0, apply_filter=True
):
    env = make_partial_observation_cheetah()
    init_state = None
    state = None
    if rnn_cell_size is not None:
        state = init_state = [np.zeros(rnn_cell_size, np.float32)]
    obs, info = env.reset()
    ep = 0
    ep_rewards = []
```

(continues on next page)

(continued from previous page)

```
    reward = 0
    while ep < n_episodes:
        if pertubation_level > 0.0:
            obs = obs + np.random.default_rng().normal(0, pertubation_level, obs.shape)

        if apply_filter:
            filter = algo.workers.local_worker().filters.get("default_policy")
            obs = filter(obs, update=False)

        if rnn_cell_size is None:
            action = algo.compute_single_action(
                obs, explore=False, policy_id="default_policy"
            )
        else:
            action, state, _ = algo.compute_single_action(
                obs, state=state, explore=False, policy_id="default_policy"
            )
        obs, r, terminated, truncated, info = env.step(action)
        reward += r
        if terminated or truncated:
            ep += 1
            obs, info = env.reset()
            state = init_state
            ep_rewards.append(reward)
            reward = 0
    return np.mean(ep_rewards)
```

Training Code

Finally, we will define a function that trains a policy network. To compare the performance of the CfC policy network with a baseline, we will make the function such that it can train a CfC policy network or an MLP baseline policy network.

```
def run_algo(model_name, num_iters):
    config = {
        "env": "my_env",
        "gamma": 0.99,
        "num_gpus": 1,
        "num_workers": 16,
        "num_envs_per_worker": 4,
        "lambda": 0.95,
        "kl_coeff": 1.0,
        "num_sgd_iter": 64,
        "lr": 0.0005,
        "vf_loss_coeff": 0.5,
        "clip_param": 0.1,
        "sgd_minibatch_size": 4096,
        "train_batch_size": 65536,
        "grad_clip": 0.5,
        "batch_mode": "truncate_episodes",
        "observation_filter": "MeanStdFilter",
```

(continues on next page)

(continued from previous page)

```
        "framework": "tf",
    }
    rnn_cell_size = None
    if model_name == "cfc_rnn":
        rnn_cell_size = 64
        config["model"] = {
            "vf_share_layers": True,
            "custom_model": "cfc_rnn",
            "custom_model_config": {
                "cell_size": rnn_cell_size,
            },
        }
    elif model_name == "default":
        pass
    else:
        raise ValueError(f"Unknown model type {model_name}")

    algo = PPO(config=config)
    history = {"reward": [], "reward_noise": [], "iteration": []}
    for iteration in range(1, num_iters + 1):
        algo.train()
        if iteration % 10 == 0 or iteration == 1:
            history["iteration"].append(iteration)
            history["reward"].append(run_closed_loop(algo, rnn_cell_size))
            history["reward_noise"].append(
                run_closed_loop(algo, rnn_cell_size, pertubation_level=0.1)
            )
            print(
                f"{model_name} iteration {iteration}: {history['reward'][-1]:0.2f}, with␣
↪noise: {history['reward_noise'][-1]:0.2f}"
            )
    return history
```

Note: Exact learning curves and performance numbers will vary between runs.

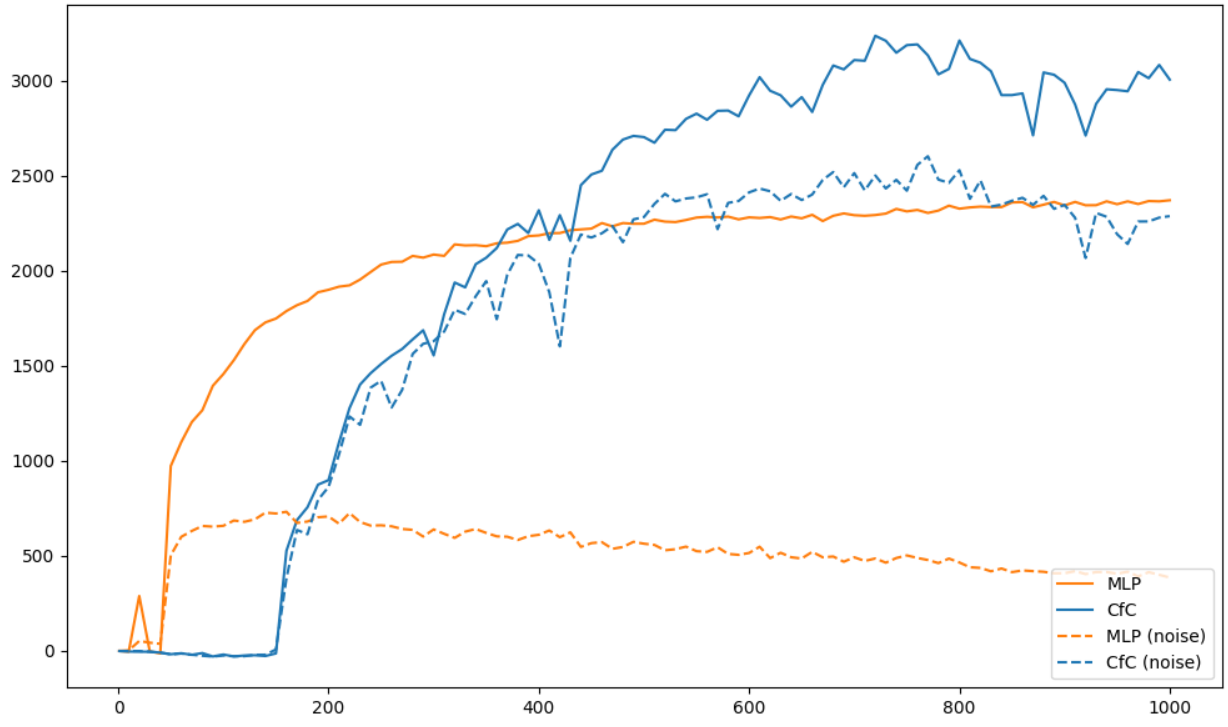
Once we have defined everything we can compare the two network architectures.

```
if __name__ == "__main__":
    ModelCatalog.register_custom_model("cfc_rnn", CustomRNN)
    register_env("my_env", lambda env_config: make_partial_observation_cheetah())
    ray.init(num_cpus=24, num_gpus=1)
    cfc_result = run_algo("cfc_rnn", 1000)
    ray.shutdown()
    ModelCatalog.register_custom_model("cfc_rnn", CustomRNN)
    register_env("my_env", lambda env_config: make_partial_observation_cheetah())
    ray.init(num_cpus=24, num_gpus=1)
    mlp_result = run_algo("default", 1000)

    fig, ax = plt.subplots(figsize=(10, 6))
    ax.plot(
        mlp_result["iteration"], mlp_result["reward"], label="MLP", color="tab:orange"
```

(continues on next page)

(continued from previous page)

```
    )
    ax.plot(
        cfc_result["iteration"], cfc_result["reward"], label="CfC", color="tab:blue"
    )
    ax.plot(
        mlp_result["iteration"],
        mlp_result["reward_noise"],
        label="MLP (noise)",
        color="tab:orange",
        ls="--",
    )
    ax.plot(
        cfc_result["iteration"],
        cfc_result["reward_noise"],
        label="CfC (noise)",
        color="tab:blue",
        ls="--",
    )
    ax.legend(loc="upper left")
    fig.tight_layout()
    plt.savefig("cfc_vs_mlp.png")
```



As we have see in the plot above, although the MLP baseline policy network experiences a faster initial improvement, the CfC policy network eventually outperforms the MLP baseline policy network, while also being more robust to noise.

1.2.4 Atari Behavior Cloning

In this guide, we will train an NCP to play Atari. Code is provided for both PyTorch and TensorFlow (toogle with the tabs). Instead of learning a policy via reinforcement learning (which can be a bit complex), we will make use of an pretrained expert policy that the NCP should copy using supervised learning (i.e., behavior cloning).

Setup and Requirements

Before we start, we need to install some packages

PyTorch

```
pip3 install ncps torch "ale-py==0.7.4" "ray[rllib]==2.1.0" "gym[atari,accept-rom-
↪license]==0.23.1"
```

TensorFlow

```
pip3 install -U ncps tensorflow "gymnasium[atari,accept-rom-license]" "ray[rllib]"
pip3 install ncps tensorflow "ale-py==0.7.4" "ray[rllib]==2.1.0" "gym[atari,accept-rom-
↪license]==0.23.1"
```

Note that this example uses older versions of `ale-py`, `ray` and `gym` due to compatibility issues with the latest versions caused by the deprecation of `gym` in favor for the `gymnasium` package.

Defining the model

First, we will define the neural network model. The model consists of a convolutional block, followed by a CfC recurrent neural network, and a final linear layer.

We first define a convolutional block that operates over just a batch of images. Each Atari image has 4 color channels and dimension of 84-by-84 pixels.

PyTorch

```
import torch.nn as nn
import torch.nn.functional as F

class ConvBlock(nn.Module):
    def __init__(self):
        super().__init__()
        self.conv1 = nn.Conv2d(4, 64, 5, padding=2, stride=2)
        self.conv2 = nn.Conv2d(64, 128, 5, padding=2, stride=2)
        self.bn2 = nn.BatchNorm2d(128)
        self.conv3 = nn.Conv2d(128, 128, 5, padding=2, stride=2)
        self.conv4 = nn.Conv2d(128, 256, 5, padding=2, stride=2)
        self.bn4 = nn.BatchNorm2d(256)
```

(continues on next page)

(continued from previous page)

```
    def forward(self, x):
        x = F.relu(self.conv1(x))
        x = F.relu(self.bn2(self.conv2(x)))
        x = F.relu(self.conv3(x))
        x = F.relu(self.bn4(self.conv4(x)))
        x = x.mean((-1, -2))  # Global average pooling
        return x
```

TensorFlow

```
import tensorflow as tf

class ConvBlock(tf.keras.models.Sequential):
def __init__(self):
    super(ConvBlock, self).__init__(
        [
            tf.keras.Input((84, 84, 4)), # batch dimension is implicit
            tf.keras.layers.Lambda(lambda x: tf.cast(x, tf.float32) / 255.0), #␣
↪normalize input
            tf.keras.layers.Conv2D(64, 5, padding="same", activation="relu", strides=2),
            tf.keras.layers.Conv2D(128, 5, padding="same", activation="relu", strides=2),
            tf.keras.layers.Conv2D(128, 5, padding="same", activation="relu", strides=2),
            tf.keras.layers.Conv2D(256, 5, padding="same", activation="relu", strides=2),
            tf.keras.layers.GlobalAveragePooling2D(),
        ]
    )
```

In PyTorch, we can use the `tensor.view()` method to reshape the input tensor. In TensorFlow, we can use the `tf.keras.layers.Reshape` layer.

Note: As pointed out by @R-Liebert Impala-style convolutional blocks are known to be more efficient than the one we use here. You can find a Tensorflow implementation of the Impala-style convolutional block here (TensorFlow).

Next, we define the full model. As the model operate over batches of sequences of images (5 dimensions), wherea the convolutional block only accepts 4-dimensional inputs, we have to reshape the input when processing it with the ConvBlock layers.

Note: In TensorFlow, we can just wrap it in a `tf.keras.layers.TimeDistributed` which takes care of exactly that. In PyTorch we can use the `tensor.view()` method.

When we apply the model in a closed-loop setting, we need some mechanisms to *remember* the hidden state, i.e., use the final hidden state of the previous data batch as the initial values of the hidden state for the current data batch. This is implemented by implementing two different inference modes of the model:

1. A training mode, where we have a single input tensor (batch of sequences of images) and predicts a single output tensor.
2. A stateful mode, where the input and output are pairs, containing the initial state of the RNN and the final state of the RNN in the input and output as second element respectively.

Note: In PyTorch we can implement this a bit cleaner by making the initial states an optional argument of the `module.forward()` method.

PyTorch

```
from ncps.torch import CfC

class ConvCfC(nn.Module):
    def __init__(self, n_actions):
        super().__init__()
        self.conv_block = ConvBlock()
        self.rnn = CfC(256, 64, batch_first=True, proj_size=n_actions)

    def forward(self, x, hx=None):
        batch_size = x.size(0)
        seq_len = x.size(1)
        # Merge time and batch dimension into a single one (because the Conv layers␣
↪require this)
        x = x.view(batch_size * seq_len, *x.shape[2:])
        x = self.conv_block(x)  # apply conv block to merged data
        # Separate time and batch dimension again
        x = x.view(batch_size, seq_len, *x.shape[1:])
        x, hx = self.rnn(x, hx)  # hx is the hidden state of the RNN
        return x, hx
```

TensorFlow

```
from ncps.tf import CfC

class ConvCfC(tf.keras.Model):
def __init__(self, n_actions):
    super().__init__()
    self.conv_block = ConvBlock()
    self.td_conv = tf.keras.layers.TimeDistributed(self.conv_block)
    self.rnn = CfC(64, return_sequences=True, return_state=True)
    self.linear = tf.keras.layers.Dense(n_actions)

def get_initial_states(self, batch_size=1):
    return self.rnn.cell.get_initial_state(batch_size=batch_size, dtype=tf.float32)

def call(self, x, training=None, **kwargs):
    has_hx = isinstance(x, list) or isinstance(x, tuple)
    initial_state = None
    if has_hx:
        # additional inputs are passed as a tuple
        x, initial_state = x

    x = self.td_conv(x, training=training)
    x, next_state = self.rnn(x, initial_state=initial_state)
```

(continues on next page)

(continued from previous page)

```
    x = self.linear(x)
    if has_hx:
        return (x, next_state)
    return x
```

Dataloader

Next, we define the Atari environment and the dataset. We have to wrap the environment with the helper functions proposed in DeepMind's Atari Nature paper, which apply the following transformations:

- Downscales the Atari frames to 84-by-84 pixels
- Converts the frames to grayscale
- Stacks 4 consecutive frames into a single observation

The resulting observations are then 84-by-84 images with 4 channels.

```
import gym
import ale_py
from ray.rllib.env.wrappers.atari_wrappers import wrap_deepmind
import numpy as np

env = gym.make("ALE/Breakout-v5")
# We need to wrap the environment with the Deepmind helper functions
env = wrap_deepmind(env)
```

For the behavior cloning dataset, we will use the `AtariCloningDataset` (PyTorch) and `AtariCloningDatasetTF` (TensorFlow) datasets provided by the `ncps` package.

PyTorch

```
import torch
from torch.utils.data import Dataset
import torch.optim as optim
from ncps.datasets.torch import AtariCloningDataset

train_ds = AtariCloningDataset("breakout", split="train")
val_ds = AtariCloningDataset("breakout", split="val")
trainloader = torch.utils.data.DataLoader(
    train_ds, batch_size=32, num_workers=4, shuffle=True
)
valloader = torch.utils.data.DataLoader(val_ds, batch_size=32, num_workers=4)
```

TensorFlow

```
from ncps.datasets.tf import AtariCloningDatasetTF

data = AtariCloningDatasetTF("breakout")
# batch size 32
trainloader = data.get_dataset(32, split="train")
valloader = data.get_dataset(32, split="val")
```

Running the model in a closed-loop

Next, we have to define the code for applying the model in a continuous control loop with the environment. There are three subtleties we need to take care of:

1. Reset the RNN hidden states when a new episode starts in the Atari game
2. Reshape the input frames to have an extra batch and time dimension of size 1 as the network accepts only batches of sequences instead of single frames
3. Pass the current hidden state together with the observation as input, and unpack the the prediction and next hidden state from the output

PyTorch

```
def run_closed_loop(model, env, num_episodes=None):
    obs = env.reset()
    device = next(model.parameters()).device
    hx = None  # Hidden state of the RNN
    returns = []
    total_reward = 0
    with torch.no_grad():
        while True:
            # PyTorch require channel first images -> transpose data
            obs = np.transpose(obs, [2, 0, 1]).astype(np.float32) / 255.0
            # add batch and time dimension (with a single element in each)
            obs = torch.from_numpy(obs).unsqueeze(0).unsqueeze(0).to(device)
            pred, hx = model(obs, hx)
            # remove time and batch dimension -> then argmax
            action = pred.squeeze(0).squeeze(0).argmax().item()
            obs, r, done, _ = env.step(action)
            total_reward += r
            if done:
                obs = env.reset()
                hx = None  # Reset hidden state of the RNN
                returns.append(total_reward)
                total_reward = 0
                if num_episodes is not None:
                    # Count down the number of episodes
                    num_episodes = num_episodes - 1
                    if num_episodes == 0:
                        return returns
```

TensorFlow

```
def run_closed_loop(model, env, num_episodes=None):
    obs = env.reset()
    hx = model.get_initial_states()
    returns = []
    total_reward = 0
    while True:
        # add batch and time dimension (with a single element in each)
        obs = np.expand_dims(np.expand_dims(obs, 0), 0)
        pred, hx = model.predict((obs, hx), verbose=0)
        action = pred[0, 0].argmax()
        # remove time and batch dimension -> then argmax
        obs, r, done, _ = env.step(action)
        total_reward += r
        if done:
            returns.append(total_reward)
            total_reward = 0
            obs = env.reset()
            hx = model.get_initial_states()
            # Reset RNN hidden states when episode is over
            if num_episodes is not None:
                # Count down the number of episodes
                num_episodes = num_episodes - 1
                if num_episodes == 0:
                    return returns
```

Training loop

PyTorch

For the training, we define a function that train the model by making one pass over the dataset. We also define an evaluation function that measure the loss and accuracy of the model.

```
def train_one_epoch(model, criterion, optimizer, trainloader):
    running_loss = 0.0
    pbar = tqdm(total=len(trainloader))
    model.train()
    device = next(model.parameters()).device  # get device the model is located on
    for i, (inputs, labels) in enumerate(trainloader):
        inputs = inputs.to(device)  # move data to same device as the model
        labels = labels.to(device)

        # zero the parameter gradients
        optimizer.zero_grad()
        # forward + backward + optimize
        outputs, hx = model(inputs)
        labels = labels.view(-1, *labels.shape[2:])  # flatten
        outputs = outputs.reshape(-1, *outputs.shape[2:])  # flatten
        loss = criterion(outputs, labels)
        loss.backward()
        optimizer.step()
```

(continues on next page)

(continued from previous page)

```
        # print statistics
        running_loss += loss.item()
        pbar.set_description(f"loss={running_loss / (i + 1):0.4g}")
        pbar.update(1)
    pbar.close()

def eval(model, valloader):
    losses, accs = [], []
    model.eval()
    device = next(model.parameters()).device  # get device the model is located on
    with torch.no_grad():
        for inputs, labels in valloader:
            inputs = inputs.to(device)  # move data to same device as the model
            labels = labels.to(device)

            outputs, _ = model(inputs)
            outputs = outputs.reshape(-1, *outputs.shape[2:])  # flatten
            labels = labels.view(-1, *labels.shape[2:])  # flatten
            loss = criterion(outputs, labels)
            acc = (outputs.argmax(-1) == labels).float().mean()
            losses.append(loss.item())
            accs.append(acc.item())
    return np.mean(losses), np.mean(accs)
```

TensorFlow

For training the model we can use the keras high-level `model.fit` functionality.

During the training with the `fit` function , we measure only offline performance in the form of the training and validation accuracy. However, we also want to check after every training epoch how the cloned network is performing when applied to the closed-loop environment. To this end, we have to define a keras callback that is invoked after every training epoch and implements the closed-loop evaluation.

```
class ClosedLoopCallback(tf.keras.callbacks.Callback):
    def __init__(self, model, env):
        super().__init__()
        self.model = model
        self.env = env

    def on_epoch_end(self, epoch, logs=None):
        r = run_closed_loop(self.model, self.env, num_episodes=10)
        print(f"\nEpoch {epoch} return: {np.mean(r):0.2f} +- {np.std(r):0.2f}")
```

Training the model

Finally, we can instantiate the model and train it.

PyTorch

```
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
model = ConvCfC(n_actions=env.action_space.n).to(device)
criterion = nn.CrossEntropyLoss()
optimizer = optim.Adam(model.parameters(), lr=0.0001)

for epoch in range(50):  # loop over the dataset multiple times
    train_one_epoch(model, criterion, optimizer, trainloader)

    # Evaluate model on the validation set
    val_loss, val_acc = eval(model, valloader)
    print(f"Epoch {epoch+1}, val_loss={val_loss:0.4g}, val_acc={100*val_acc:0.2f}%")

    # Apply model in closed-loop environment
    returns = run_closed_loop(model, env, num_episodes=10)
    print(f"Mean return {np.mean(returns)} (n={len(returns)})")
```

TensorFlow

```
model = ConvCfC(env.action_space.n)

model.compile(
    loss=tf.keras.losses.SparseCategoricalCrossentropy(from_logits=True),
    optimizer=tf.keras.optimizers.Adam(0.0001),
    metrics=[tf.keras.metrics.SparseCategoricalAccuracy()],
)
# (batch, time, height, width, channels)
model.build((None, None, 84, 84, 4))
model.summary()

model.fit(
    trainloader,
    epochs=50,
    validation_data=valloader,
    callbacks=[
        ClosedLoopCallback(model, env)
    ],
)
```

After the training is completed we can display in a window how the model plays the game.

```
# Visualize Atari game and play endlessly
env = gym.make("ALE/Breakout-v5", render_mode="human")
env = wrap_deepmind(env)
run_closed_loop(model, env)
```

The full source code can be downloaded here (PyTorch) and here (TensorFlow).

Note: At a validation accuracy of about 92%, the behavior cloning data usually implies a decent closed-loop performance of the cloned agent.

The output of the full script is something like:

PyTorch

```
> loss=0.4349: 100%|| 938/938 [01:35<00:00,  9.83it/s]
> Epoch 1, val_loss=1.67, val_acc=31.94%
> Mean return 0.2 (n=10)
> loss=0.2806: 100%|| 938/938 [01:30<00:00, 10.33it/s]
> Epoch 2, val_loss=0.43, val_acc=83.51%
> Mean return 3.7 (n=10)
> loss=0.223: 100%|| 938/938 [01:31<00:00, 10.30it/s]
> Epoch 3, val_loss=0.2349, val_acc=91.43%
> Mean return 4.9 (n=10)
> loss=0.1951: 100%|| 938/938 [01:31<00:00, 10.26it/s]
> Epoch 4, val_loss=2.824, val_acc=29.19%
> Mean return 0.6 (n=10)
> loss=0.1786: 100%|| 938/938 [01:30<00:00, 10.33it/s]
> Epoch 5, val_loss=0.3122, val_acc=89.03%
> Mean return 4.0 (n=10)
> loss=0.1669: 100%|| 938/938 [01:31<00:00, 10.22it/s]
> Epoch 6, val_loss=4.272, val_acc=22.84%
> Mean return 0.5 (n=10)
> loss=0.1575: 100%|| 938/938 [01:32<00:00, 10.14it/s]
> Epoch 7, val_loss=0.2788, val_acc=89.78%
> Mean return 9.9 (n=10)
> loss=0.15: 100%|| 938/938 [01:33<00:00, 10.08it/s]
> Epoch 8, val_loss=3.725, val_acc=25.07%
> Mean return 0.6 (n=10)
> loss=0.1429: 100%|| 938/938 [01:31<00:00, 10.23it/s]
> Epoch 9, val_loss=0.5851, val_acc=77.82%
> Mean return 44.6 (n=10)
> loss=0.1369: 100%|| 938/938 [01:32<00:00, 10.12it/s]
> Epoch 10, val_loss=0.7148, val_acc=71.74%
> Mean return 3.4 (n=10)
> loss=0.1316: 100%|| 938/938 [01:32<00:00, 10.11it/s]
> Epoch 11, val_loss=0.2138, val_acc=92.27%
> Mean return 15.8 (n=10)
> loss=0.1267: 100%|| 938/938 [01:33<00:00, 10.02it/s]
> Epoch 12, val_loss=0.2683, val_acc=90.54%
> Mean return 14.3 (n=10)
> ....
```

TensorFlow

```
> Model: "sequential_1"
> _________________________________________________________________
>  Layer (type)                 Output Shape              Param #
> =================================================================
>  time_distributed (TimeDistr  (None, None, 256)        1440576
>  ibuted)
>
>  cf_c (CfC)                   (None, None, 64)          74112
>
>  dense (Dense)                (None, None, 4)           260
>
> =================================================================
> Total params: 1,514,948
> Trainable params: 1,514,948
> Non-trainable params: 0
> _________________________________________________________________
> Epoch 1/50
> 2022-10-11 15:45:55.524895: I tensorflow/stream_executor/cuda/cuda_dnn.cc:384] Loaded␣
↪cuDNN version 8302
> 2022-10-11 15:45:56.037075: I tensorflow/core/platform/default/subprocess.cc:304]␣
↪Start cannot spawn child process: No such file or directory
> 938/938 [==============================] - ETA: 0s - loss: 0.4964 - sparse_categorical_
↪accuracy: 0.8305
> Epoch 0 return: 2.50 +- 1.91
> 938/938 [==============================] - 413s 436ms/step - loss: 0.4964 - sparse_
↪categorical_accuracy: 0.8305 - val_loss: 0.3931 - val_sparse_categorical_accuracy: 0.
↪8633
> Epoch 2/50
> 938/938 [==============================] - ETA: 0s - loss: 0.3521 - sparse_categorical_
↪accuracy: 0.8752
> Epoch 1 return: 4.00 +- 3.58
> 938/938 [==============================] - 450s 480ms/step - loss: 0.3521 - sparse_
↪categorical_accuracy: 0.8752 - val_loss: 0.3168 - val_sparse_categorical_accuracy: 0.
↪8884
> Epoch 3/50
> 938/938 [==============================] - ETA: 0s - loss: 0.3009 - sparse_categorical_
↪accuracy: 0.8918
> Epoch 2 return: 5.30 +- 3.32
> 938/938 [==============================] - 469s 501ms/step - loss: 0.3009 - sparse_
↪categorical_accuracy: 0.8918 - val_loss: 0.2732 - val_sparse_categorical_accuracy: 0.
↪9020
> Epoch 4/50
> 938/938 [==============================] - ETA: 0s - loss: 0.2690 - sparse_categorical_
↪accuracy: 0.9029
> Epoch 3 return: 13.90 +- 9.54
> 938/938 [==============================] - 514s 548ms/step - loss: 0.2690 - sparse_
↪categorical_accuracy: 0.9029 - val_loss: 0.2485 - val_sparse_categorical_accuracy: 0.
↪9103
> Epoch 5/50
> 938/938 [==============================] - ETA: 0s - loss: 0.2501 - sparse_categorical_
↪accuracy: 0.9095
```

(continues on next page)

(continued from previous page)

```
> Epoch 4 return: 15.50 +- 14.33
> 938/938 [==============================] - 516s 550ms/step - loss: 0.2501 - sparse_
↪categorical_accuracy: 0.9095 - val_loss: 0.2475 - val_sparse_categorical_accuracy: 0.
↪9107
> Epoch 6/50
> 938/938 [==============================] - ETA: 0s - loss: 0.2361 - sparse_categorical_
↪accuracy: 0.9145
> Epoch 5 return: 16.00 +- 12.49
> 938/938 [==============================] - 514s 548ms/step - loss: 0.2361 - sparse_
↪categorical_accuracy: 0.9145 - val_loss: 0.2363 - val_sparse_categorical_accuracy: 0.
↪9150
> Epoch 7/50
> 938/938 [==============================] - ETA: 0s - loss: 0.2257 - sparse_categorical_
↪accuracy: 0.9184
> Epoch 6 return: 35.60 +- 30.20
> 938/938 [==============================] - 508s 542ms/step - loss: 0.2257 - sparse_
↪categorical_accuracy: 0.9184 - val_loss: 0.2256 - val_sparse_categorical_accuracy: 0.
↪9183
> Epoch 8/50
> 938/938 [==============================] - ETA: 0s - loss: 0.2173 - sparse_categorical_
↪accuracy: 0.9213
> Epoch 7 return: 7.70 +- 5.59
> 938/938 [==============================] - 501s 534ms/step - loss: 0.2173 - sparse_
↪categorical_accuracy: 0.9213 - val_loss: 0.2179 - val_sparse_categorical_accuracy: 0.
↪9207
> Epoch 9/50
> 938/938 [==============================] - ETA: 0s - loss: 0.2095 - sparse_categorical_
↪accuracy: 0.9239
> Epoch 8 return: 67.40 +- 81.60
> 938/938 [==============================] - 555s 592ms/step - loss: 0.2095 - sparse_
↪categorical_accuracy: 0.9239 - val_loss: 0.2045 - val_sparse_categorical_accuracy: 0.
↪9265
> Epoch 10/50
> 938/938 [==============================] - ETA: 0s - loss: 0.2032 - sparse_categorical_
↪accuracy: 0.9263
> Epoch 9 return: 15.20 +- 12.16
> 938/938 [==============================] - 523s 558ms/step - loss: 0.2032 - sparse_
↪categorical_accuracy: 0.9263 - val_loss: 0.1962 - val_sparse_categorical_accuracy: 0.
↪9290
> Epoch 11/50
> 938/938 [==============================] - ETA: 0s - loss: 0.1983 - sparse_categorical_
↪accuracy: 0.9279
> Epoch 10 return: 26.50 +- 27.98
> 938/938 [==============================] - 512s 546ms/step - loss: 0.1983 - sparse_
↪categorical_accuracy: 0.9279 - val_loss: 0.2180 - val_sparse_categorical_accuracy: 0.
↪9210
> Epoch 12/50
> 938/938 [==============================] - ETA: 0s - loss: 0.1926 - sparse_categorical_
↪accuracy: 0.9302
> Epoch 11 return: 53.00 +- 79.22
> 938/938 [==============================] - 1846s 2s/step - loss: 0.1926 - sparse_
↪categorical_accuracy: 0.9302 - val_loss: 0.1924 - val_sparse_categorical_accuracy: 0.
```

(continues on next page)

(continued from previous page)

```
↪9311
> ....
```

1.2.5 Atari Reinforcement Learning (PPO)

In this guide, we will train an NCP to play Atari through Reinforcement Learning. Code is provided for TensorFlow and relies on ray[rllib] for the learning. The specific RL algorithm we are using is proximal policy optimization (PPO) which is a good baseline that works for both discrete and continuous action space environments.

Setup and Requirements

Before we start, we need to install some packages

```
pip3 install ncps tensorflow "ale-py==0.7.4" "ray[rllib]==2.1.0" "gym[atari,accept-rom-
↪license]==0.23.1"
```

Defining the model

First, we will define the neural network model. The model consists of a convolutional block, followed by a CfC recurrent neural network. To make our model compatible with rllib we have to subclass from `ray.rllib.models.tf.recurrent_net.RecurrentNetwork`.

Our Conv-CfC network has two output tensors:

- A distribution of the possible actions (= the policy)
- A scalar estimation of the value function

The second output is a necessity of the PPO RL algorithms we are using. Learning both, the policy and the value function, in a single network often has some learning advantages of having shared features.

```
import numpy as np
from ray.rllib.models.modelv2 import ModelV2
from ray.rllib.models.tf.recurrent_net import RecurrentNetwork
from ray.rllib.utils.annotations import override
import tensorflow as tf
from ncps.tf import CfC

class ConvCfCModel(RecurrentNetwork):
    """Example of using the Keras functional API to define a RNN model."""

    def __init__(
        self,
        obs_space,
        action_space,
        num_outputs,
        model_config,
        name,
        cell_size=64,
    ):
```

(continues on next page)

(continued from previous page)

```
        super(ConvCfCModel, self).__init__(
            obs_space, action_space, num_outputs, model_config, name
        )
        self.cell_size = cell_size

        # Define input layers
        input_layer = tf.keras.layers.Input(
            # rllib flattens the input
            shape=(None, obs_space.shape[0] * obs_space.shape[1] * obs_space.shape[2]),
            name="inputs",
        )
        state_in_h = tf.keras.layers.Input(shape=(cell_size,), name="h")
        seq_in = tf.keras.layers.Input(shape=(), name="seq_in", dtype=tf.int32)

        # Preprocess observation with a hidden layer and send to CfC
        self.conv_block = tf.keras.models.Sequential([
            tf.keras.Input(
                (obs_space.shape[0] * obs_spac.shapee[1] * obs_space.shape[2])
            ),  # batch dimension is implicit
            tf.keras.layers.Lambda(
                lambda x: tf.cast(x, tf.float32) / 255.0
            ),  # normalize input
            # unflatten the input image that has been done by rllib
            tf.keras.layers.Reshape((obs_space.shape[0], obs_space.shape[1], obs_space.
↪shape[2])),
            tf.keras.layers.Conv2D(
                64, 5, padding="same", activation="relu", strides=2
            ),
            tf.keras.layers.Conv2D(
                128, 5, padding="same", activation="relu", strides=2
            ),
            tf.keras.layers.Conv2D(
                128, 5, padding="same", activation="relu", strides=2
            ),
            tf.keras.layers.Conv2D(
                256, 5, padding="same", activation="relu", strides=2
            ),
            tf.keras.layers.GlobalAveragePooling2D(),
        ])
        self.td_conv = tf.keras.layers.TimeDistributed(self.conv_block)

        dense1 = self.td_conv(input_layer)
        cfc_out, state_h = CfC(
            cell_size, return_sequences=True, return_state=True, name="cfc"
        )(
            inputs=dense1,
            mask=tf.sequence_mask(seq_in),
            initial_state=[state_in_h],
        )

        # Postprocess CfC output with another hidden layer and compute values
        logits = tf.keras.layers.Dense(
```

(continues on next page)

(continued from previous page)

```
            self.num_outputs, activation=tf.keras.activations.linear, name="logits"
        )(cfc_out)
        values = tf.keras.layers.Dense(1, activation=None, name="values")(cfc_out)

        # Create the RNN model
        self.rnn_model = tf.keras.Model(
            inputs=[input_layer, seq_in, state_in_h],
            outputs=[logits, values, state_h],
        )
        self.rnn_model.summary()

    @override(RecurrentNetwork)
    def forward_rnn(self, inputs, state, seq_lens):
        model_out, self._value_out, h = self.rnn_model([inputs, seq_lens] + state)
        return model_out, [h]

    @override(ModelV2)
    def get_initial_state(self):
        return [
            np.zeros(self.cell_size, np.float32),
        ]

    @override(ModelV2)
    def value_function(self):
        return tf.reshape(self._value_out, [-1])
```

Once we have defined out model, we can register it in rllib:

```
from ray.rllib.models import ModelCatalog

ModelCatalog.register_custom_model("cfc", ConvCfCModel)
```

Defining the RL algorithm and its hyperparameters

Every RL algorithm relies on dozen of hyperparameters that can have a huge effect on the learning performance. PPO is no exception to this rule. Luckily, the rllib authors have provided a configuration that works decently for PPO with Atari environments, which we will make use of.

```
import argparse
import os
import gym
from ray.tune.registry import register_env
from ray.rllib.algorithms.ppo import PPO
import time
import ale_py
from ray.rllib.env.wrappers.atari_wrappers import wrap_deepmind

if __name__ == "__main__":
    parser = argparse.ArgumentParser()
    parser.add_argument("--env", type=str, default="ALE/Breakout-v5")
    parser.add_argument("--cont", default="")
```

(continues on next page)

(continued from previous page)

```
    parser.add_argument("--render", action="store_true")
    parser.add_argument("--hours", default=4, type=int)
    args = parser.parse_args()

    register_env("atari_env", lambda env_config: wrap_deepmind(gym.make(args.env)))
    config = {
        "env": "atari_env",
        "preprocessor_pref": None,
        "gamma": 0.99,
        "num_gpus": 1,
        "num_workers": 16,
        "num_envs_per_worker": 4,
        "create_env_on_driver": True,
        "lambda": 0.95,
        "kl_coeff": 0.5,
        "clip_rewards": True,
        "clip_param": 0.1,
        "vf_clip_param": 10.0,
        "entropy_coeff": 0.01,
        "rollout_fragment_length": 100,
        "sgd_minibatch_size": 500,
        "num_sgd_iter": 10,
        "batch_mode": "truncate_episodes",
        "observation_filter": "NoFilter",
        "model": {
            "vf_share_layers": True,
            "custom_model": "cfc",
            "max_seq_len": 20,
            "custom_model_config": {
                "cell_size": 64,
            },
        },
        "framework": "tf2",
    }

    algo = PPO(config=config)
```

When running the algorithm, we will create checkpoints which we can restore later on. We will store these checkpoints in the folder `rl_ckpt` and add the ability to restore a specific checkpoint id via the `--cont` argument.

```
os.makedirs(f"rl_ckpt/{args.env}", exist_ok=True)
if args.cont != "":
    algo.load_checkpoint(f"rl_ckpt/{args.env}/checkpoint-{args.cont}")
```

Visualizing the policy-environment interaction

To later on visualize how the trained policy is playing the Atari game, we have to write a function that enables the `render_mode` of the environment and executes the policy in a closed-loop.

For computing the actions we use the `compute_single_action` function of the algorithm object, but we have to take care of the RNN hidden state initialization ourselves.

```
def run_closed_loop(algo, config):
    env = gym.make(args.env, render_mode="human")
    env = wrap_deepmind(env)
    rnn_cell_size = config["model"]["custom_model_config"]["cell_size"]
    obs = env.reset()
    state = init_state = [np.zeros(rnn_cell_size, np.float32)]
    while True:
        action, state, _ = algo.compute_single_action(
            obs, state=state, explore=False, policy_id="default_policy"
        )
        obs, reward, done, _ = env.step(action)
        if done:
            obs = env.reset()
            state = init_state
```

Running PPO

Finally, we can run the RL algorithm. Particularly, we branch depending on the `--render` argument whether to train the policy or visualize it.

```
if args.render:
    run_closed_loop(
        algo,
        config,
    )
else:
    start_time = time.time()
    last_eval = 0
    while True:
        info = algo.train()
        if time.time() - last_eval > 60 * 5:  # every 5 minutes print some stats
            print(f"Ran {(time.time()-start_time)/60/60:0.1f} hours")
            print(
                f"    sampled {info['info']['num_env_steps_sampled']/1000:0.0f}k steps"
            )
            print(f"    policy reward: {info['episode_reward_mean']:0.1f}")
            last_eval = time.time()
            ckpt = algo.save_checkpoint(f"rl_ckpt/{args.env}")
            print(f"    saved checkpoint '{ckpt}'")

        elapsed = (time.time() - start_time) / 60  # in minutes
        if elapsed > args.hours * 60:
            break
```

The full source code can be found here.

Note: On a modern desktop machine, it takes about an hour to get to a return of 20, and about 4 hours to reach a return of 50.

Warning: For Atari environments rllib distinguishes between the episodic (1 life) and the game (3 lives) return, thus the return reported by rllib may differ from the return achieved in the closed-loop evaluation.

The output of the full script is something like:

```
> Ran 0.0 hours
>     sampled 4k steps
>     policy reward: nan
>     saved checkpoint 'rl_ckpt/ALE/Breakout-v5/checkpoint-1'
> Ran 0.1 hours
>     sampled 52k steps
>     policy reward: 1.9
>     saved checkpoint 'rl_ckpt/ALE/Breakout-v5/checkpoint-13'
> Ran 0.2 hours
>     sampled 105k steps
>     policy reward: 2.6
>     saved checkpoint 'rl_ckpt/ALE/Breakout-v5/checkpoint-26'
> Ran 0.3 hours
>     sampled 157k steps
>     policy reward: 3.4
>     saved checkpoint 'rl_ckpt/ALE/Breakout-v5/checkpoint-39'
> Ran 0.4 hours
>     sampled 210k steps
>     policy reward: 6.7
>     saved checkpoint 'rl_ckpt/ALE/Breakout-v5/checkpoint-52'
> Ran 0.4 hours
>     sampled 266k steps
>     policy reward: 8.7
>     saved checkpoint 'rl_ckpt/ALE/Breakout-v5/checkpoint-66'
> Ran 0.5 hours
>     sampled 323k steps
>     policy reward: 10.5
>     saved checkpoint 'rl_ckpt/ALE/Breakout-v5/checkpoint-80'
> Ran 0.6 hours
>     sampled 379k steps
>     policy reward: 10.7
>     saved checkpoint 'rl_ckpt/ALE/Breakout-v5/checkpoint-94'
...
```

1.3 API Reference

1.3.1 PyTorch (nn.modules)

Sequence models

class ncps.torch.**CfC**(*input_size*, *units*, *proj_size=None*, *return_sequences=True*, *batch_first=True*, *mixed_memory=False*, *mode='default'*, *activation='lecun_tanh'*, *backbone_units=None*, *backbone_layers=None*, *backbone_dropout=None*)

Bases: `Module`

Applies a Closed-form Continuous-time RNN to an input sequence.

Examples:

```
>>> from ncps.torch import CfC
>>>
>>> rnn = CfC(20,50)
>>> x = torch.randn(2, 3, 20) # (batch, time, features)
>>> h0 = torch.zeros(2,50) # (batch, units)
>>> output, hn = rnn(x,h0)
```

Parameters

- **input_size** (*Union[int, Wiring]*) – Number of input features
- **units** – Number of hidden units
- **proj_size** (*Optional[int]*) – If not None, the output of the RNN will be projected to a tensor with dimension proj_size (i.e., an implict linear output layer)
- **return_sequences** (*bool*) – Whether to return the full sequence or just the last output
- **batch_first** (*bool*) – Whether the batch or time dimension is the first (0-th) dimension
- **mixed_memory** (*bool*) – Whether to augment the RNN by a memory-cell to help learn long-term dependencies in the data
- **mode** (*str*) – Either "default", "pure" (direct solution approximation), or "no_gate" (without second gate).
- **activation** (*str*) – Activation function used in the backbone layers
- **backbone_units** (*Optional[int]*) – Number of hidden units in the backbone layer (default 128)
- **backbone_layers** (*Optional[int]*) – Number of backbone layers (default 1)
- **backbone_dropout** (*Optional[int]*) – Dropout rate in the backbone layers (default 0)

training: **bool**

forward(*input*, *hx=None*, *timespans=None*)

Parameters

- **input** – Input tensor of shape (L,C) in batchless mode, or (B,L,C) if batch_first was set to True and (L,B,C) if batch_first is False

- **hx** – Initial hidden state of the RNN of shape (B,H) if mixed_memory is False and a tuple ((B,H),(B,H)) if mixed_memory is True. If None, the hidden states are initialized with all zeros.
- **timespans** –

Returns

A pair (output, hx), where output and hx the final hidden state of the RNN

class ncps.torch.**LTC**(*input_size*, *units*, *return_sequences=True*, *batch_first=True*, *mixed_memory=False*, *input_mapping='affine'*, *output_mapping='affine'*, *ode_unfolds=6*, *epsilon=1e-08*, *implicit_param_constraints=True*)

Bases: `Module`

Applies a Liquid time-constant (LTC) RNN to an input sequence.

Examples:

```
>>> from ncps.torch import LTC
>>>
>>> rnn = LTC(20,50)
>>> x = torch.randn(2, 3, 20) # (batch, time, features)
>>> h0 = torch.zeros(2,50) # (batch, units)
>>> output, hn = rnn(x,h0)
```

Note: For creating a wired Neural circuit policy (NCP) you can pass a *ncps.wirings.NCP* object instead of the number of units

---

Examples:

```
>>> from ncps.torch import LTC
>>> from ncps.wirings import NCP
>>>
>>> wiring = NCP(10, 10, 8, 6, 6, 4, 6)
>>> rnn = LTC(20, wiring)

>>> x = torch.randn(2, 3, 20) # (batch, time, features)
>>> h0 = torch.zeros(2, 28) # (batch, units)
>>> output, hn = rnn(x,h0)
```

Parameters

- **input_size** (*int*) – Number of input features
- **units** – Wiring (ncps.wirings.Wiring instance) or integer representing the number of (fully-connected) hidden units
- **return_sequences** (*bool*) – Whether to return the full sequence or just the last output
- **batch_first** (*bool*) – Whether the batch or time dimension is the first (0-th) dimension
- **mixed_memory** (*bool*) – Whether to augment the RNN by a memory-cell to help learn long-term dependencies in the data
- **input_mapping** –
- **output_mapping** –
- **ode_unfolds** –

- **epsilon** –
- **implicit_param_constraints** –

property state_size

property sensory_size

property motor_size

property output_size

property synapse_count

property sensory_synapse_count

forward(*input*, *hx=None*, *timespans=None*)

Parameters

- **input** – Input tensor of shape (L,C) in batchless mode, or (B,L,C) if batch_first was set to True and (L,B,C) if batch_first is False
- **hx** – Initial hidden state of the RNN of shape (B,H) if mixed_memory is False and a tuple ((B,H),(B,H)) if mixed_memory is True. If None, the hidden states are initialized with all zeros.
- **timespans** –

Returns
A pair (output, hx), where output and hx the final hidden state of the RNN

training: bool

Single time-step models (RNN cells)

class ncps.torch.**CfCCell**(*input_size*, *hidden_size*, *mode='default'*, *backbone_activation='lecun_tanh'*, *backbone_units=128*, *backbone_layers=1*, *backbone_dropout=0.0*, *sparsity_mask=None*)

Bases: Module

A Closed-form Continuous-time cell.

Note: This is an RNNCell that process single time-steps. To get a full RNN that can process sequences see *ncps.torch.CfC*.

Parameters

- **input_size** –
- **hidden_size** –
- **mode** –
- **backbone_activation** –
- **backbone_units** –
- **backbone_layers** –

- **backbone_dropout** –
- **sparsity_mask** –

init_weights()

forward(*input*, *hx*, *ts*)

training: bool

class ncps.torch.**LTCCell**(*wiring*, *in_features=None*, *input_mapping='affine'*, *output_mapping='affine'*, *ode_unfolds=6*, *epsilon=1e-08*, *implicit_param_constraints=False*)

Bases: Module

A Liquid time-constant (LTC) cell.

Note: This is an RNNCell that process single time-steps. To get a full RNN that can process sequences see *ncps.torch.LTC*.

Parameters

- **wiring** –
- **in_features** –
- **input_mapping** –
- **output_mapping** –
- **ode_unfolds** –
- **epsilon** –
- **implicit_param_constraints** –

property state_size

property sensory_size

property motor_size

property output_size

property synapse_count

property sensory_synapse_count

add_weight(*name*, *init_value*, *requires_grad=True*)

apply_weight_constraints()

forward(*inputs*, *states*, *elapsed_time=1.0*)

training: bool

1.3.2 Tensorflow (tf.keras Layers)

Sequence models

class ncps.tf.**CfC**(**args*, ***kwargs*)

Bases: RNN

Applies a Closed-form Continuous-time RNN to an input sequence.

Examples:

```
>>> from ncps.tf import CfC
>>>
>>> rnn = CfC(50)
>>> x = tf.random.uniform((2, 10, 20))  # (B,L,C)
>>> y = rnn(x)
```

Parameters

- **units** – Number of hidden units
- **mixed_memory** – Whether to augment the RNN by a memory-cell to help learn long-term dependencies in the data (default False)
- **mode** – Either "default", "pure" (direct solution approximation), or "no_gate" (without second gate). (default "default)
- **activation** – Activation function used in the backbone layers (default "lecun_tanh")
- **backbone_units** – Number of hidden units in the backbone layer (default 128)
- **backbone_layers** – Number of backbone layers (default 1)
- **backbone_dropout** – Dropout rate in the backbone layers (default 0)
- **return_sequences** – Whether to return the full sequence or just the last output (default False)
- **return_state** – Whether to return just the output of the RNN or a tuple (output, last_hidden_state) (default False)
- **go_backwards** – If True, the input sequence will be process from back to the front (default False)
- **stateful** – Whether to remember the last hidden state of the previous inference/training batch and use it as initial state for the next inference/training batch (default False)
- **unroll** – Whether to unroll the graph, i.e., may increase speed at the cost of more memory (default False)
- **time_major** – Whether the time or batch dimension is the first (0-th) dimension (default False)
- **kwargs** –

class ncps.tf.**LTC**(**args*, ***kwargs*)

Bases: RNN

Applies a Liquid time-constant (LTC) RNN to an input sequence.

Examples:

```
>>> from ncps.tf import LTC
>>>
>>> rnn = LTC(50)
>>> x = tf.random.uniform((2, 10, 20))  # (B,L,C)
>>> y = rnn(x)
```

Note: For creating a wired Neural circuit policy (NCP) you can pass a *ncps.wirings.NCP* object instead of the number of units

Examples:

```
>>> from ncps.tf import LTC
>>> from ncps.wirings import NCP
>>>
>>> wiring = NCP(10, 10, 8, 6, 6, 4, 4)
>>> rnn = LTC(wiring)
>>> x = tf.random.uniform((2, 10, 20))  # (B,L,C)
>>> y = rnn(x)
```

Parameters

- **units** – Wiring (ncps.wirings.Wiring instance) or integer representing the number of (fully-connected) hidden units
- **mixed_memory** – Whether to augment the RNN by a memory-cell to help learn long-term dependencies in the data
- **input_mapping** – Mapping applied to the sensory neurons. Possible values None, “linear”, “affine” (default “affine”)
- **output_mapping** – Mapping applied to the motor neurons. Possible values None, “linear”, “affine” (default “affine”)
- **ode_unfolds** – Number of ODE-solver steps per time-step (default 6)
- **epsilon** – Auxillary value to avoid dividing by 0 (default 1e-8)
- **initialization_ranges** – A dictionary for overwriting the range of the uniform weight initialization (default None)
- **return_sequences** – Whether to return the full sequence or just the last output (default False)
- **return_state** – Whether to return just the output of the RNN or a tuple (output, last_hidden_state) (default False)
- **go_backwards** – If True, the input sequence will be process from back to the front (default False)
- **stateful** – Whether to remember the last hidden state of the previous inference/training batch and use it as initial state for the next inference/training batch (default False)
- **unroll** – Whether to unroll the graph, i.e., may increase speed at the cost of more memory (default False)
- **time_major** – Whether the time or batch dimension is the first (0-th) dimension (default False)
- **kwargs** –

Single time-step models (RNN cells)

class ncps.tf.**CfCCell**(**args*, ***kwargs*)

Bases: AbstractRNNCell

A Closed-form Continuous-time cell.

Note: This is an RNNCell that process single time-steps. To get a full RNN that can process sequences, see *ncps.tf.CfC* or wrap the cell with a tf.keras.layers.RNN.

Parameters

- **units** – Number of hidden units
- **input_sparsity** –
- **recurrent_sparsity** –
- **mode** – Either "default", "pure" (direct solution approximation), or "no_gate" (without second gate).
- **activation** – Activation function used in the backbone layers
- **backbone_units** – Number of hidden units in the backbone layer (default 128)
- **backbone_layers** – Number of backbone layers (default 1)
- **backbone_dropout** – Dropout rate in the backbone layers (default 0)
- **kwargs** –

property **state_size**

build(*input_shape*)

call(*inputs*, *states*, ***kwargs*)

class ncps.tf.**LTCCell**(**args*, ***kwargs*)

Bases: AbstractRNNCell

A Liquid time-constant (LTC) cell.

Note: This is an RNNCell that process single time-steps. To get a full RNN that can process sequences, see *ncps.tf.LTC* or wrap the cell with a tf.keras.layers.RNN.

Examples:

```
>>> import ncps
>>> from ncps.tf import LTCCell
>>>
>>> wiring = ncps.wirings.Random(16, output_dim=2, sparsity_level=0.5)
>>> cell = LTCCell(wiring)
>>> rnn = tf.keras.layers.RNN(cell)
>>> x = tf.random.uniform((1,4)) # (batch, features)
>>> h0 = tf.zeros((1, 16))
>>> y = cell(x,h0)
>>>
```

(continues on next page)

(continued from previous page)

```
>>> x_seq = tf.random.uniform((1,20,4)) # (batch, time, features)
>>> y_seq = rnn(x_seq)
```

Parameters

- **wiring** –
- **input_mapping** –
- **output_mapping** –
- **ode_unfolds** –
- **epsilon** –
- **initialization_ranges** –
- **kwargs** –

property state_size

property sensory_size

property motor_size

property output_size

build(*input_shape*)

call(*inputs*, *states*)

get_config()

classmethod from_config(*config*)

1.3.3 Wirings

class ncps.wirings.**Wiring**(*units*)

Bases: object

property num_layers

get_neurons_of_layer(*layer_id*)

is_built()

build(*input_dim*)

erev_initializer(*shape=None*, *dtype=None*)

sensory_erev_initializer(*shape=None*, *dtype=None*)

set_input_dim(*input_dim*)

set_output_dim(*output_dim*)

get_type_of_neuron(*neuron_id*)

add_synapse(*src*, *dest*, *polarity*)

add_sensory_synapse(*src*, *dest*, *polarity*)

get_config()

classmethod from_config(*config*)

get_graph(*include_sensory_neurons=True*)

Returns a networkx.DiGraph object of the wiring diagram :param include_sensory_neurons: Whether to include the sensory neurons as nodes in the graph

property synapse_count

Counts the number of synapses between internal neurons of the model

property sensory_synapse_count

Counts the number of synapses from the inputs (sensory neurons) to the internal neurons of the model

draw_graph(*layout='shell'*, *neuron_colors=None*, *synapse_colors=None*, *draw_labels=False*)

Draws a matplotlib graph of the wiring structure Examples:

```
>>> import matplotlib.pyplot as plt
>>> plt.figure(figsize=(6, 4))
>>> legend_handles = wiring.draw_graph(draw_labels=True)
>>> plt.legend(handles=legend_handles, loc="upper center", bbox_to_anchor=(1,␣
↪1))
>>> plt.tight_layout()
>>> plt.show()
```

Parameters

- **layout** –
- **neuron_colors** –
- **synapse_colors** –
- **draw_labels** –

Returns

class ncps.wirings.**AutoNCP**(*units*, *output_size*, *sparsity_level=0.5*, *seed=22222*)

Bases: *NCP*

Instantiate an NCP wiring with only needing to specify the number of units and the number of outputs

Parameters

- **units** – The total number of neurons
- **output_size** – The number of motor neurons (=output size). This value must be less than units-2 (typically good choices are 0.3 times the total number of units)
- **sparsity_level** – A hyperparameter between 0.0 (very dense) and 0.9 (very sparse) NCP.
- **seed** – Random seed for generating the wiring

class ncps.wirings.**NCP**(*inter_neurons*, *command_neurons*, *motor_neurons*, *sensory_fanout*, *inter_fanout*, *recurrent_command_synapses*, *motor_fanin*, *seed=22222*)

Bases: *Wiring*

Creates a Neural Circuit Policies wiring. The total number of neurons (= state size of the RNN) is given by the sum of inter, command, and motor neurons. For an easier way to generate a NCP wiring see the `AutoNCP` wiring class.

Parameters

- **inter_neurons** – The number of inter neurons (layer 2)
- **command_neurons** – The number of command neurons (layer 3)
- **motor_neurons** – The number of motor neurons (layer 4 = number of outputs)
- **sensory_fanout** – The average number of outgoing synapses from the sensory to the inter neurons
- **inter_fanout** – The average number of outgoing synapses from the inter to the command neurons
- **recurrent_command_synapses** – The average number of recurrent connections in the command neuron layer
- **motor_fanin** – The average number of incoming synapses of the motor neurons from the command neurons
- **seed** – The random seed used to generate the wiring

property num_layers

get_neurons_of_layer(*layer_id*)

get_type_of_neuron(*neuron_id*)

build(*input_shape*)

class ncps.wirings.**FullyConnected**(*units*, *output_dim=None*, *erev_init_seed=1111*, *self_connections=True*)

Bases: *Wiring*

build(*input_shape*)

class ncps.wirings.**Random**(*units*, *output_dim=None*, *sparsity_level=0.0*, *random_seed=1111*)

Bases: *Wiring*

build(*input_shape*)

PYTHON MODULE INDEX

n

INDEX

O

R

S

T

W