agentMET4FOF Documentation

Bang Xiang Yong

Apr 29, 2020

Getting started:

agentMET4FOF is a Python library developed at the Institute for Manufacturing of the University of Cambridge (UK) as part of the European joint Research Project EMPIR 17IND12 Met4FoF.

For the *agentMET4FOF* homepage go to GitHub.

agentMET4FOF is written in Python 3.

CircleCI Documentation Status Codecov Badge

CHAPTER 1

Multi-Agent System for Metrology for Factory of the Future (Met4FoF) Code

This is supported by European Metrology Programme for Innovation and Research (EMPIR) under the project Metrology for the Factory of the Future (Met4FoF), project number 17IND12. (https://www.ptb.de/empir2018/met4fof/home/)

1.1 About

- How can metrological input be incorporated into an agent-based system for addressing uncertainty of machine learning in future manufacturing?
- Includes agent-based simulation and implementation
- Readthedocs documentation is available at (https://agentmet4fof.readthedocs.io)

1.2 Use agentMET4FOF

The easiest way to get started with *agentMET4FOF* is navigating to the folder in which you want to create a virtual Python environment (*venv*), create one based on Python 3.6 or later, activate it, first install numpy, then install *agentMET4FOF* from PyPI.org and then work through the tutorials or examples. To do this, issue the following commands on your Shell:

```
$ cd /LOCAL/PATH/TO/ENVS
$ python3 -m venv agentMET4FOF_venv
$ source agentMET4FOF_venv/bin/activate
(agentMET4FOF_venv) $ pip install numpy
Collecting numpy
...
Successfully installed numpy-...
(agentMET4FOF_venv) $ pip install agentMET4FOF
```

(continues on next page)

(continued from previous page)

```
Collecting agentMET4FOF
...
Successfully installed agentMET4FOF-... ...
(agentMET4FOF_venv) $ python
Python ... (default, ..., ...)
[GCC ...] on ...
Type "help", "copyright", "credits" or "license" for more information.
>>> from agentMET4FOF_tutorials import tutorial_1_generator_agent
>>> tutorial_1_generator_agent.demonstrate_generator_agent_use()
Starting NameServer...
Broadcast server running on 0.0.0.0:9091
NS running on 127.0.0.1:3333 (127.0.0.1)
URI = PYRO:Pyro.NameServer@127.0.0.1:3333
INFO [2020-02-21 19:04:26.961014] (AgentController): INITIALIZED
INFO [2020-02-21 19:04:27.032258] (Logger): INITIALIZED
 * Serving Flask app "agentMET4FOF.dashboard.Dashboard" (lazy loading)
 * Environment: production
   WARNING: This is a development server. Do not use it in a production deployment.
   Use a production WSGI server instead.
 * Debug mode: off
 * Running on http://127.0.0.1:8050/ (Press CTRL+C to quit)
...
```

Now you can visit `http://127.0.0.1:8050/` with any Browser and watch the SineGenerator agent you just spawned.

To get some insights and really get going please visit agentMET4FOF.readthedocs.io .

1.3 Get started developing

First clone the repository to your local machine as described here. To get started with your present *Anaconda* installation just go to *Anaconda prompt*, navigate to your local clone

```
cd /LOCAL/PATH/TO/agentMET4FOF
```

and execute

```
conda env create --file environment.yml
```

This will create an *Anaconda* virtual environment with all dependencies satisfied. If you don't have *Anaconda* installed already follow this guide first, then create the virtual environment as stated above and then proceed.

Alternatively, for non-conda environments, you can install the dependencies using pip

```
pip install -r requirements.txt
```

First take a look at the tutorials and examples or start hacking if you already are familiar with agentMET4FOF and want to customize your agents' network.

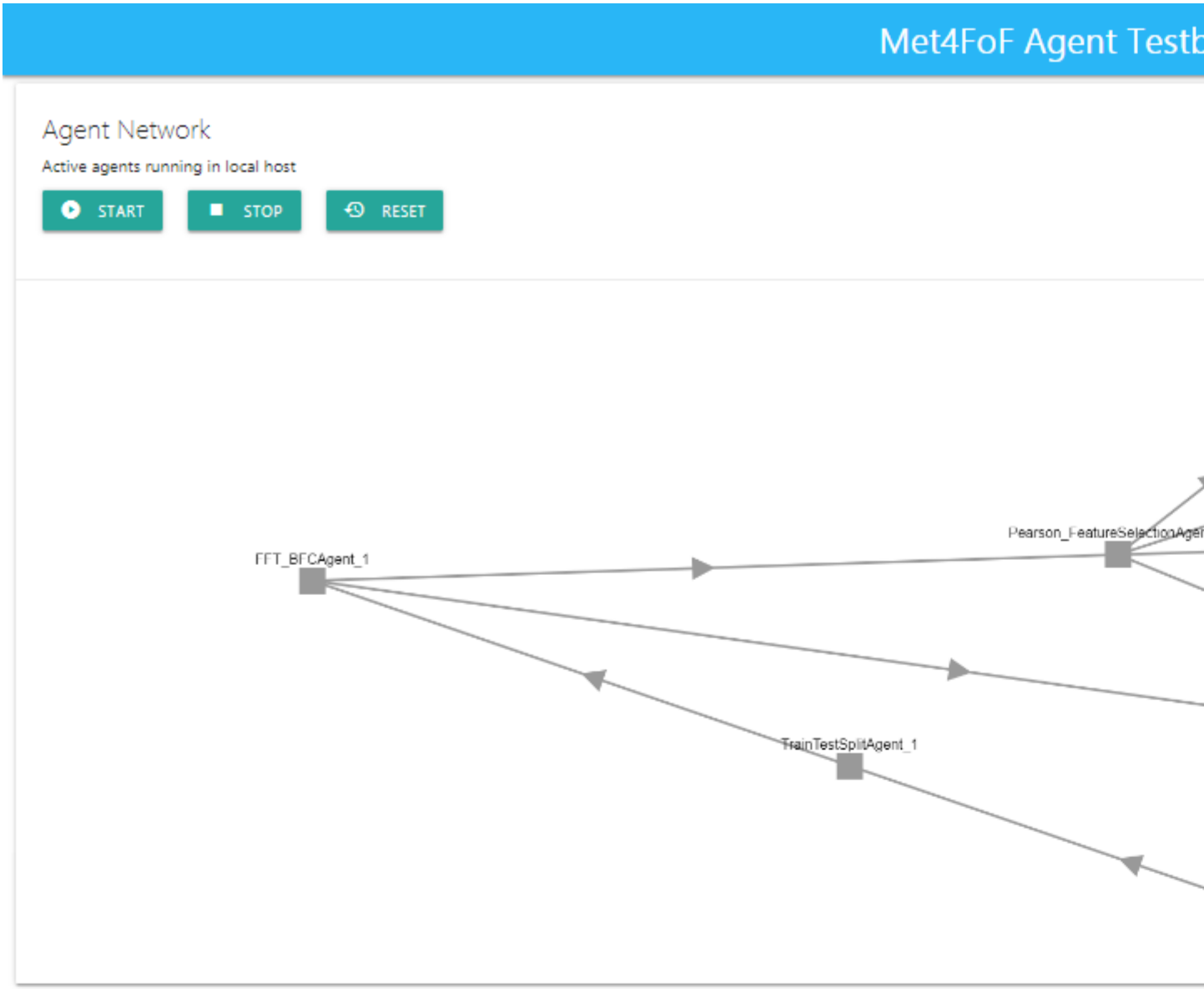
Alternatively, watch the tutorial webinar here

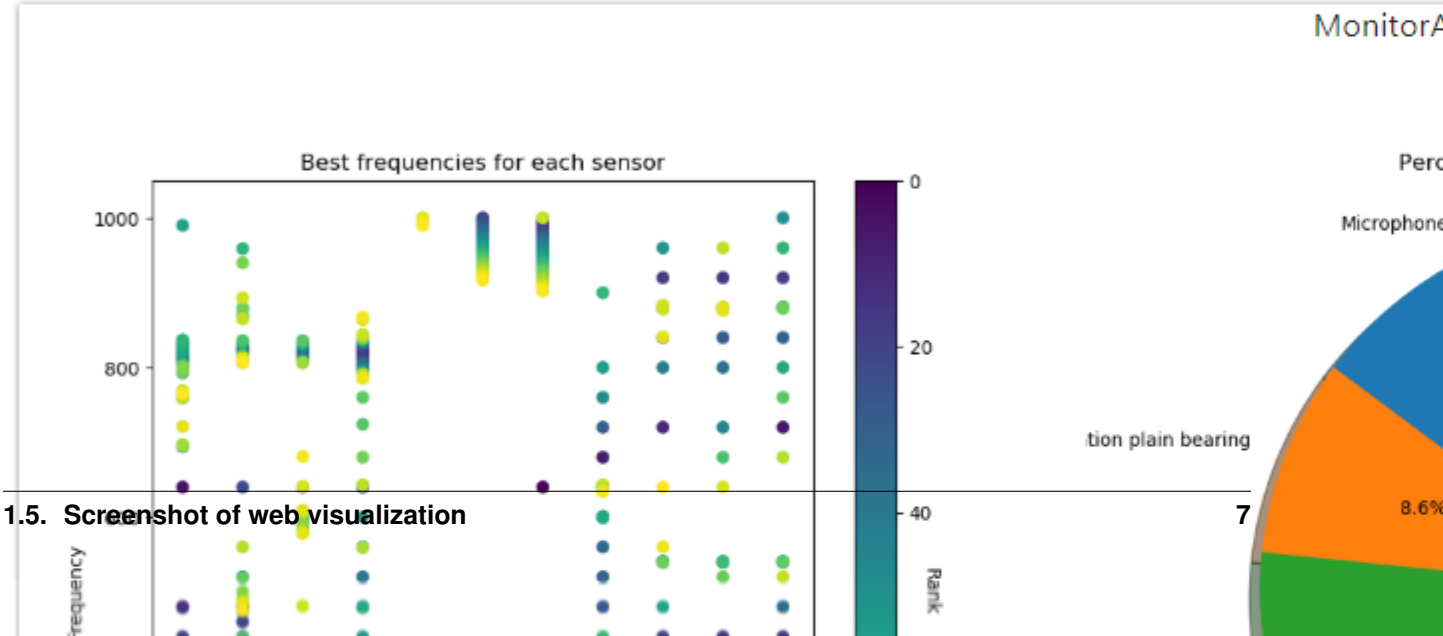
1.4 Updates

- Implemented base class AgentMET4FOF with built-in agent classes DataStreamAgent, MonitorAgent

- Implemented class AgentNetwork to start or connect to a agent server
- Implemented with ZEMA prognosis of Electromechanical cylinder data set as use case DOI
- Implemented interactive web application with user interface

1.5 Screenshot of web visualization





1.6 Note

- In the event of agents not terminating cleanly, run

```
taskkill /f /im python.exe /t
```

  in Windows Command Prompt to terminate all background python processes.

CHAPTER 2

Tutorial 1 - A simple pipeline to plot a signal

First we define a simple pipeline of two agents, of which one will generate a signal (in our case a *SineGeneratorAgent*) and the other one plots the signal on the dashboard (this is always a *MonitorAgent*).

We define a *SineGeneratorAgent* for which we have to override the functions `init_parameters()` & `agent_loop()` to define the new agent's behaviour.

- `init_parameters()` is used to setup the input data stream and potentially other necessary parameters.
- `agent_loop()` will be endlessly repeated until further notice. It will sample by sample extract the input data stream's content and push it to all agents connected to *SineGeneratorAgent*'s output channel by invoking `send_output()`.

The *MonitorAgent* is connected to the *SineGeneratorAgent*'s output channel and per default automatically plots the output.

Each agent has an internal `current_state` which can be used as a switch to change the behaviour of the agent. The available states are listed here.

As soon as all agents are initialized and the connections are set up, the agent network is started by accordingly changing all agents' state simultaneously.

[1]:
```
# %load tutorial_1_generator_agent.py
from agentMET4FOF.agents import AgentMET4FOF, AgentNetwork, MonitorAgent
from agentMET4FOF.streams import SineGenerator


class SineGeneratorAgent(AgentMET4FOF):
    """An agent streaming a sine signal

    Takes samples from the :py:mod:`SineGenerator` and pushes them sample by sample
    to connected agents via its output channel.
    """
    _sine_stream: SineGenerator

    def init_parameters(self):
        """Initialize the input data
```

(continues on next page)

(continued from previous page)

```
        Initialize the input data stream as an instance of the
        :py:mod:`SineGenerator` class
        """
        self._sine_stream = SineGenerator()

    def agent_loop(self):
        """Model the agent's behaviour

        On state *Running* the agent will extract sample by sample the input data
        streams content and push it via invoking :py:method:`AgentMET4FOF.send_output`.
        """
        if self.current_state == "Running":
            sine_data = self._sine_stream.next_sample()  # dictionary
            self.send_output(sine_data["x"])


def demonstrate_generator_agent_use():
    # Start agent network server.
    agent_network = AgentNetwork()

    # Initialize agents by adding them to the agent network.
    gen_agent = agent_network.add_agent(agentType=SineGeneratorAgent)
    monitor_agent = agent_network.add_agent(agentType=MonitorAgent)

    # Interconnect agents by either way:
    # 1) by agent network.bind_agents(source, target).
    agent_network.bind_agents(gen_agent, monitor_agent)

    # 2) by the agent.bind_output().
    gen_agent.bind_output(monitor_agent)

    # Set all agents' states to "Running".
    agent_network.set_running_state()

    # Allow for shutting down the network after execution
    return agent_network


if __name__ == "__main__":
    demonstrate_generator_agent_use()
```

```
Starting NameServer...
Broadcast server running on 0.0.0.0:9091
NS running on 127.0.0.1:3333 (127.0.0.1)
URI = PYRO:Pyro.NameServer@127.0.0.1:3333
INFO [2020-04-24 09:21:23.002156] (AgentController): INITIALIZED
INFO [2020-04-24 09:21:23.200110] (SineGeneratorAgent_1): INITIALIZED
 * Serving Flask app "agentMET4FOF.dashboard.Dashboard" (lazy loading)
 * Environment: production
   WARNING: This is a development server. Do not use it in a production deployment.
   Use a production WSGI server instead.
 * Debug mode: off
INFO [2020-04-24 09:21:23.294149] (MonitorAgent_1): INITIALIZED
[2020-04-24 09:21:23.360214] (SineGeneratorAgent_1): Connected output module:␣
↪MonitorAgent_1
```

(continues on next page)

(continued from previous page)

```
SET STATE:   Running
[2020-04-24 09:21:24.218709] (SineGeneratorAgent_1): Pack time: 0.000897
[2020-04-24 09:21:24.223207] (SineGeneratorAgent_1): Sending: [0.]
[2020-04-24 09:21:24.225895] (MonitorAgent_1): Received: {'from': 'SineGeneratorAgent_
↪1', 'data': array([0.]), 'senderType': 'SineGeneratorAgent', 'channel': 'default'}
[2020-04-24 09:21:24.226708] (MonitorAgent_1): Tproc: 0.00017
[2020-04-24 09:21:25.217698] (SineGeneratorAgent_1): Pack time: 0.000419
[2020-04-24 09:21:25.220045] (SineGeneratorAgent_1): Sending: [0.47942554]
[2020-04-24 09:21:25.221406] (MonitorAgent_1): Received: {'from': 'SineGeneratorAgent_
↪1', 'data': array([0.47942554]), 'senderType': 'SineGeneratorAgent', 'channel':
↪'default'}
[2020-04-24 09:21:25.223795] (MonitorAgent_1): Memory: {'SineGeneratorAgent_1':␣
↪array([0.        , 0.47942554])}
[2020-04-24 09:21:25.224696] (MonitorAgent_1): Tproc: 0.00264
[2020-04-24 09:21:26.217785] (SineGeneratorAgent_1): Pack time: 0.000415
[2020-04-24 09:21:26.218995] (SineGeneratorAgent_1): Sending: [0.84147098]
[2020-04-24 09:21:26.220107] (MonitorAgent_1): Received: {'from': 'SineGeneratorAgent_
↪1', 'data': array([0.84147098]), 'senderType': 'SineGeneratorAgent', 'channel':
↪'default'}
[2020-04-24 09:21:26.222038] (MonitorAgent_1): Memory: {'SineGeneratorAgent_1':␣
↪array([0.        , 0.47942554, 0.84147098])}
[2020-04-24 09:21:26.223223] (MonitorAgent_1): Tproc: 0.002481
[2020-04-24 09:21:27.216058] (SineGeneratorAgent_1): Pack time: 0.000131
[2020-04-24 09:21:27.216323] (SineGeneratorAgent_1): Sending: [0.99749499]
[2020-04-24 09:21:27.216876] (MonitorAgent_1): Received: {'from': 'SineGeneratorAgent_
↪1', 'data': array([0.99749499]), 'senderType': 'SineGeneratorAgent', 'channel':
↪'default'}
[2020-04-24 09:21:27.217220] (MonitorAgent_1): Memory: {'SineGeneratorAgent_1':␣
↪array([0.        , 0.47942554, 0.84147098, 0.99749499])}
[2020-04-24 09:21:27.217288] (MonitorAgent_1): Tproc: 0.000314
[2020-04-24 09:21:28.215905] (SineGeneratorAgent_1): Pack time: 0.000102
[2020-04-24 09:21:28.216367] (MonitorAgent_1): Received: {'from': 'SineGeneratorAgent_
↪1', 'data': array([0.90929743]), 'senderType': 'SineGeneratorAgent', 'channel':
↪'default'}
[2020-04-24 09:21:28.216186] (SineGeneratorAgent_1): Sending: [0.90929743]
[2020-04-24 09:21:28.216623] (MonitorAgent_1): Memory: {'SineGeneratorAgent_1':␣
↪array([0.        , 0.47942554, 0.84147098, 0.99749499, 0.90929743])}
[2020-04-24 09:21:28.216678] (MonitorAgent_1): Tproc: 0.000229
```

```
 * Running on http://127.0.0.1:8050/ (Press CTRL+C to quit)
```

```
<Figure size 432x288 with 0 Axes>
```

CHAPTER 3

Tutorial 2 - A simple pipeline with signal postprocessing.

Here we demonstrate how to build a *MathAgent* as an intermediate to process the *SineGeneratorAgent*'s output before plotting. Subsequently, a *MultiMathAgent* is built to show the ability to send a dictionary of multiple fields to the recipient. We provide a custom `on_received_message()` function, which is called every time a message is received from input agents.

The received message is a dictionary of the form:

```
{
'from':agent_name,
'data': data,
'senderType': agent_class_name,
'channel':'channel_name'
}
```

By default, `'channel'` is set to `"default"`, however a custom channel can be set when needed, which is demonstrated in the next tutorial.

[1]:
```
# %load tutorial_2_math_agent.py
from agentMET4FOF.agents import AgentMET4FOF, AgentNetwork, MonitorAgent
from agentMET4FOF.streams import SineGenerator


# Define simple math functions.
def divide_by_two(numerator: float) -> float:
    return numerator / 2


def minus(minuend: float, subtrahend: float) -> float:
    return minuend - subtrahend


def plus(summand_1: float, summand_2: float) -> float:
    return summand_1+summand_2
```

(continues on next page)

(continued from previous page)

```
class MathAgent(AgentMET4FOF):
    def on_received_message(self, message):
        data = divide_by_two(message['data'])
        self.send_output(data)

class MultiMathAgent(AgentMET4FOF):
    def init_parameters(self,minus_param=0.5,plus_param=0.5):
        self.minus_param = minus_param
        self.plus_param = plus_param

    def on_received_message(self, message):
        minus_data = minus(message['data'], self.minus_param)
        plus_data = plus(message['data'], self.plus_param)

        self.send_output({'minus':minus_data,'plus':plus_data})

class SineGeneratorAgent(AgentMET4FOF):
    def init_parameters(self):
        self.stream = SineGenerator()

    def agent_loop(self):
        if self.current_state == "Running":
            sine_data = self.stream.next_sample() #dictionary
            self.send_output(sine_data['x'])


def main():
    # start agent network server
    agentNetwork = AgentNetwork()
    # init agents
    gen_agent = agentNetwork.add_agent(agentType=SineGeneratorAgent)
    math_agent = agentNetwork.add_agent(agentType=MathAgent)
    multi_math_agent = agentNetwork.add_agent(agentType=MultiMathAgent)
    monitor_agent = agentNetwork.add_agent(agentType=MonitorAgent)
    # connect agents : We can connect multiple agents to any particular agent
    agentNetwork.bind_agents(gen_agent, math_agent)
    agentNetwork.bind_agents(gen_agent, multi_math_agent)
    # connect
    agentNetwork.bind_agents(gen_agent, monitor_agent)
    agentNetwork.bind_agents(math_agent, monitor_agent)
    agentNetwork.bind_agents(multi_math_agent, monitor_agent)
    # set all agents states to "Running"
    agentNetwork.set_running_state()

    # allow for shutting down the network after execution
    return agentNetwork


if __name__ == '__main__':
    main()
```

```
---------------------------------------------------------------------------
ModuleNotFoundError                       Traceback (most recent call last)
```

(continues on next page)

(continued from previous page)

```
<ipython-input-1-e91ecde3d431> in <module>
      1 # %load tutorial_2_math_agent.py
----> 2 from agentMET4FOF.agents import AgentMET4FOF, AgentNetwork, MonitorAgent
      3 from agentMET4FOF.streams import SineGenerator
      4
      5

ModuleNotFoundError: No module named 'agentMET4FOF'
```

CHAPTER 4

Tutorial 3 - An advanced pipeline with multichannel signals.

We can use different channels for the receiver to handle specifically each channel name. This can be useful for example in splitting train and test channels in machine learning Then, the user will need to implement specific handling of each channel in the receiving agent.

In this example, the *MultiGeneratorAgent* is used to send two different types of data - Sine and Cosine generator. This is done via specifying `send_output (channel="sine")` and `send_output(channel="cosine")`.

Then on the receiving end, the `on_received_message()` function checks for `message['channel']` to handle it separately.

Note that by default, *MonitorAgent* is only subscribed to the `"default"` channel. Hence it will not respond to the `"cosine"` and `"sine"` channel.

[1]:
```
# %load tutorial_3_multi_channel.py
from agentMET4FOF.agents import AgentMET4FOF, AgentNetwork, MonitorAgent
from agentMET4FOF.streams import SineGenerator, CosineGenerator


def minus(data, minus_val):
    return data-minus_val


def plus(data,plus_val):
    return data+plus_val


class MultiGeneratorAgent(AgentMET4FOF):
    def init_parameters(self):
        self.sine_stream = SineGenerator()
        self.cos_stream = CosineGenerator()

    def agent_loop(self):
        if self.current_state == "Running":
            sine_data = self.sine_stream.next_sample() #dictionary
            cosine_data = self.sine_stream.next_sample() #dictionary
```

(continues on next page)

(continued from previous page)

```
            self.send_output(sine_data['x'], channel="sine")
            self.send_output(cosine_data['x'], channel="cosine")


class MultiOutputMathAgent(AgentMET4FOF):
    def init_parameters(self,minus_param=0.5,plus_param=0.5):
        self.minus_param = minus_param
        self.plus_param = plus_param

    def on_received_message(self, message):
        """
        Checks for message['channel'] and handles them separately
        Acceptable channels are "cosine" and "sine"
        """
        if message['channel'] == "cosine":
            minus_data = minus(message['data'], self.minus_param)
            self.send_output({'cosine_minus':minus_data})
        elif message['channel'] == 'sine':
            plus_data = plus(message['data'], self.plus_param)
            self.send_output({'sine_plus':plus_data})


def main():
    # start agent network server
    agentNetwork = AgentNetwork()
    # init agents
    gen_agent = agentNetwork.add_agent(agentType=MultiGeneratorAgent)
    multi_math_agent = agentNetwork.add_agent(agentType=MultiOutputMathAgent)
    monitor_agent = agentNetwork.add_agent(agentType=MonitorAgent)
    # connect agents : We can connect multiple agents to any particular agent
    # However the agent needs to implement handling multiple inputs
    agentNetwork.bind_agents(gen_agent, multi_math_agent)
    agentNetwork.bind_agents(gen_agent, monitor_agent)
    agentNetwork.bind_agents(multi_math_agent, monitor_agent)
    # set all agents states to "Running"
    agentNetwork.set_running_state()

    # allow for shutting down the network after execution
    return agentNetwork


if __name__ == '__main__':
    main()
```

```
---------------------------------------------------------------------------
ModuleNotFoundError                       Traceback (most recent call last)
<ipython-input-1-90d955fea48f> in <module>
      1 # %load tutorial_3_multi_channel.py
----> 2 from agentMET4FOF.agents import AgentMET4FOF, AgentNetwork, MonitorAgent
      3 from agentMET4FOF.streams import SineGenerator, CosineGenerator
      4
      5
```

(continues on next page)

(continued from previous page)

```
ModuleNotFoundError: No module named 'agentMET4FOF'
```

CHAPTER 5

agentMET4FOF agents

CHAPTER 6

agentMET4FOF streams

class agentMET4FOF.streams.**CosineGenerator**(*num_cycles=1000*)

class agentMET4FOF.streams.**DataStreamMET4FOF**

Class for creating finite datastream for ML with *x* as inputs and *y* as target Data can be fetched sequentially using *next_sample()* or all at once *all_samples()*

For sensors data: The format shape for 2D data stream (num_samples, n_sensors) The format shape for 3D data stream (num_samples, sample_length , n_sensors)

all_samples()

Returns all the samples in the data stream

Returns samples

Return type dict of the form *{'x': current_sample_x, 'y': current_sample_y}*

next_sample(*batch_size=1*)

Fetches the samples from the data stream and advances the internal pointer *current_idx*

Parameters **batch_size** (*int*) – number of batches to get from data stream

Returns samples

Return type dict of the form *{'x': current_sample_x, 'y': current_sample_y}*

class agentMET4FOF.streams.**SineGenerator**(*num_cycles=1000*)

agentMET4FOF.streams.**extract_x_y**(*message*)

Extracts features & target from *message['data']* with expected structure such as :

1. tuple - (x,y)
2. dict - {'x':x_data,'y':y_data}

Handle data structures of dictionary to extract features & target

CHAPTER 7

Indices and tables

- genindex
- modindex
- search

CHAPTER 8

References

Bibliography

[Bang2019] Bang X. Yong, A. Brintrup Multi Agent System for Machine Learning Under Uncertainty in Cyber Physical Manufacturing System, 9th Workshop on Service Oriented, Holonic and Multi-agent Manufacturing Systems for Industry of the Future

Python Module Index

a

Index