FOSSProF Final Report
Implementing & Integrating uncertainty-aware machine learning methods in UQpy

Project Overview

The project extends UQpy, a general-purpose Python package for uncertainty quantification, to implement uncertainty-aware scientific machine learning methods. A new Scientific Machine Learning module is introduced in UQpy using the Pytorch library (Version 2.2.1) as a backend. This module is focused on quantifying uncertainties for multiple existing neural networks and neural operator architectures via established methods such as variational inference and probabilistic dropout. The module allows plug-and-play integration with the existing modules of Pytorch.

With the surge in the use of scientific machine learning methods across disciplines, quantifying uncertainties in the predictions of data-driven models becomes critical. This module provides easy and user-friendly integration of uncertainty quantification capabilities to existing network architectures, benefiting a large number of users within the scientific machine learning community. The module can be accessed from the publicly available GitHub repository:
https://github.com/SURGroup/UQpy/tree/feature/scientific_machine_learning
The code will be fully integrated into the master branch of UQpy in the next month, triggering a new official release of the software.

Project Activities and Progress

The scientific machine learning module is implemented in four different levels of functionality:

- Layers: These define the computations that serve as the primary building blocks for various neural network architectures. Instances of UQpy layers can be integrated with PyTorch layers to construct neural networks, enabling seamless integration between PyTorch's existing neural network functionalities and the additional capabilities introduced by UQpy.
- Neural Networks: These define global architecture types for different varieties of neural networks. Neural network classes handle the attributes required for training different architectures and provide flexibility to develop new architectures.
- Loss Functions: These define the objective functions or part of the objective functions necessary for training the neural networks. For example, while training a Bayesian neural network using variational inference, the divergence between the prior and variational posterior distributions is necessary to compute the ELBO loss. This divergence term can be computed using the UQpy loss functions.
- Trainers: These handle the various steps necessary to train a neural network, such as data processing, optimizing, sampling in the case of a Bayesian neural network, etc. Trainers combine all the other modules to learn the parameters of the neural networks to perform a specific task.

The following common methods in UQ and scientific machine learning have been implemented in the UQpy module:

- Layers
    - Probabilistic Dropout Layers: These layers implement the probabilistic dropout method to train a Bayesian neural network
    - Bayesian Layers: These layers are building blocks to train a Bayesian neural network using the variational inference approach.
    - Fourier layers: These implement the basic operations necessary to construct a Fourier network
- Loss Functions

    - Kullback-Liebler divergence: Computes the KL divergence between prior and the variational posterior distributions of the parameters in a Bayesian neural network
    - Jensen Shannon divergence: Computes the JS divergence between prior and the variational posterior distributions of the parameters in a Bayesian neural network
    - Lp norm loss: Computes the Lp norm between two vectors
- Neural Networks: The following varieties of neural network architectures are implemented
    - Feed-forward networks
    - U-Nets
    - Deep Operator Networks
    - Bayesian Neural Networks
- Trainers
    - Deterministic training: Trainer to learn the parameters of a deterministic network
    - Bayes by Backprop training: Trainer to learn the posterior distributions of a Bayesian neural network using variational inference.

In addition to these, base classes are defined for each of these methods for efficient future developments. Documentation, unit tests, and integration tests have been written for all implemented features. The introduction of the scientific machine learning module has given a new direction to the UQpy software package with significant additions to the code base. The scientific machine learning module has increased the code base of UQpy by 80%. One of the key challenges in developing this module was to manage its interactions with existing modules of UQpy and the Pytorch library. It was essential to ensure seamless integration between these modules to enable plug-and-play compatibility with existing ones. This was achieved by thoughtfully designing the structure of the scientific machine learning module, prioritizing data type compatibility, and efficient communication pathways between various modules.

Outcomes and Impact

The scientific machine learning module of UQpy adds to the existing body of open-source software for implementing various neural network architectures in Python. Further, the module adds a new direction by enabling uncertainty quantification capabilities for scientific machine learning. This will help advance the application of uncertainty quantification techniques in machine learning models.

Community engagement: The scientific machine learning module of UQpy is presented in two workshops to engage with the machine learning and uncertainty quantification communities. The details of the workshops are as follows:

- Mini-tutorial "Multi-Modal Data Driven and Physics-Informed Machine Learning with Uncertainty for Materials Applications", SIAM Conference on Materials Science (MS24) held at Carnegie Mellon University in Pittsburgh, PA from May 19-23 2024. Presented by Michael Shields.
- Software tutorial at the Workshop on Frontiers of Uncertainty Quantification held at the Technical University of Braunschweig, Germany from September 24-27 2024, Braunschweig, Germany. Presented by Connor Krill.

With the rapid developments in the field of scientific machine learning and uncertainty quantification, the future goals for the project would be to integrate new methods and techniques into the existing framework. The module is designed in a way that allows for the flexibility of extending the modules to keep pace with the developments in the community.

In addition, a journal paper on the UQpy Scientific Machine learning module in currently in development. This paper, entitled “Version 4.2 - Plug and Play Uncertainty Quantification Scientific Machine Learning in UQpy,” will be published as a Software Update paper in the journal *SoftwareX,* following on our prior

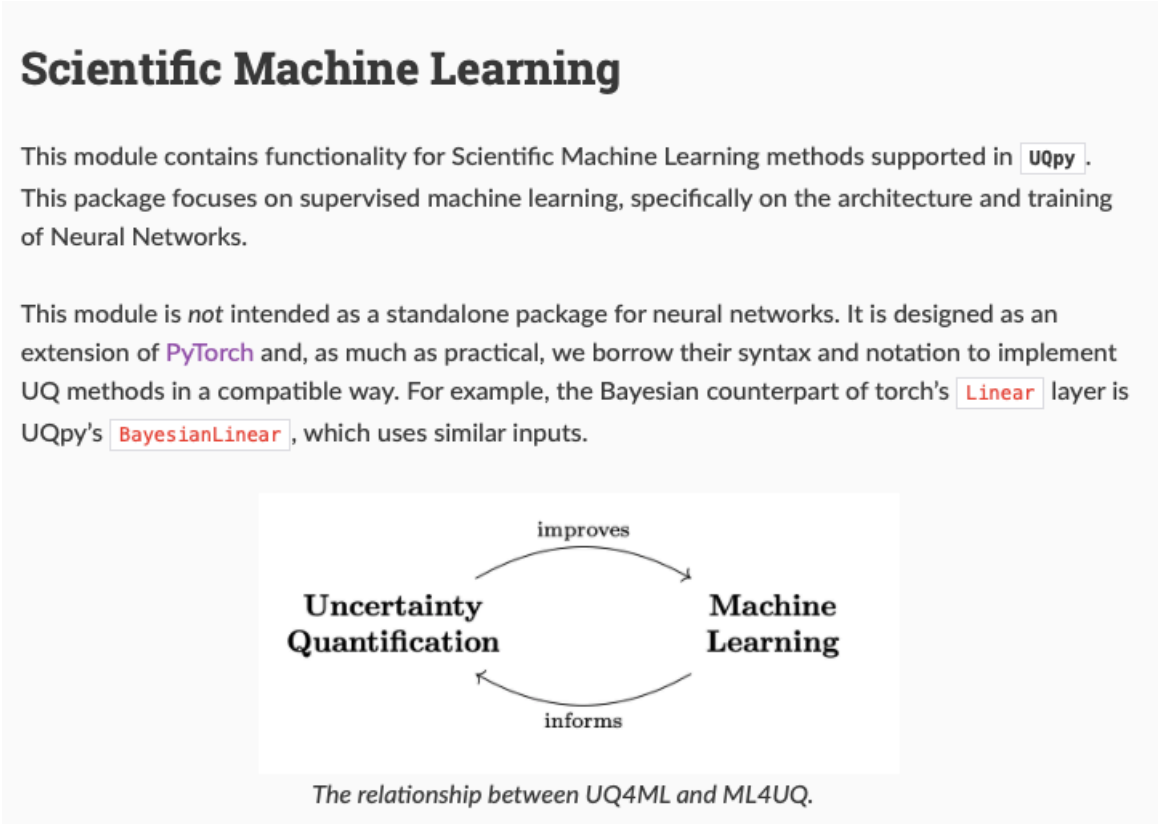
submission on the core code in reference [1]. We expect submission of the journal article in the next month.

Upon release, all updates will be supported with documentation on the UQpy documentation page at the link provided below:

A screenshot of this pending documentation is shown in the attachments below.

References:
[1] Tsapetis, D., Shields, M. D., Giovanis, D. G., Olivier, A., Novak, L., Chakroborty, P., ... & Gardner, M. (2023). Uqpy v4. 1: Uncertainty quantification with python. *SoftwareX*, *24*, 101561.

Attachments:
The following is a screenshot of a page of documentation and a code sample from UQpy's Scientific machine-learning module:



Attachment 1: A sample documentation page from UQpy

```python
import torch
import torch.nn as nn
from torchvision import datasets
from torchvision.transforms import ToTensor
import matplotlib.pyplot as plt
import UQpy.scientific_machine_learning as sml

plt.style.use("ggplot")


class BayesianNeuralNetwork(nn.Module):
    """UQpy: Replace torch's nn.Linear with UQpy's sml.BayesianLinear"""

    def __init__(self):
        super().__init__()
        self.flatten = nn.Flatten()
        self.linear_relu_stack = nn.Sequential(
            sml.BayesianLinear(28 * 28, 512),  # nn.Linear(28 * 28, 512)
            nn.ReLU(),
            sml.BayesianLinear(512, 512),  # nn.Linear(512, 512)
            nn.ReLU(),
            sml.BayesianLinear(512, 512),  # nn.Linear(512, 512)
            nn.ReLU(),
            sml.BayesianLinear(512, 10),  # nn.Linear(512, 10)
        )

    def forward(self, x):
        x = self.flatten(x)
        logits = self.linear_relu_stack(x)
        return logits
```

Attachment 2: A sample code from UQpy examples