train_deterministic_nn

September 24, 2024

1 UQpy at FrontUQ 2024

1.1 Deterministic Neural Network

This notebook trains a deterministic neural network as a surrogate for the aerodynamic model defined by XFoil. This notebook uses PyTorch and UQpy's Scientific Machine Learning module to define and train the neural network. An outline of this notebook is below.

1. Generate the training data using UM-Bridge calls to the XFoil model
2. Define and train a deterministic neural network
3. Plot the results

First, we import the necessary packages.

[1]:
```
import torch
import torch.nn as nn
import UQpy as uq
import UQpy.scientific_machine_learning as sml
import umbridge

# import logging  # Optional, display UQpy logs
# logger = logging.getLogger("UQpy")
# logger.setLevel(logging.INFO)
```

1.2 1. Compute training and testing data

We use PyTorch's Dataset and DataLoader classes to define the training and testing data for the neural network. These are convenient ways to define the datasets and, as we'll see later on, are compatible with UQpy's training infrastructure.

Note that the `AerodynamicDataset` rescales all the input variables to be within the range $[0, 1]$. The neural network is trained on scaled data, and the outputs will be restored to their original ranges before being plotted.

This class is identical to the `AerodynamicDataset` in `train_bayesian_nn.ipynb`.

[2]:
```
class AerodynamicDataset(torch.utils.data.Dataset):
    def __init__(self, n: int):
        """Construct a dataset with ``n`` samples from the XFoil model
```

```
        :param n: Total number of samples in the dataset.
        """
        self.n = n
        # define inputs using UQ
        marginals = [
            uq.Normal(0.0, 0.1),
            uq.Normal(500_000, 2_500),
            uq.Normal(0.3, 0.015),
            uq.Normal(0.7, 0.021),
            uq.Normal(0, 0.08),
        ]
        distribution = uq.JointIndependent(marginals)
        x = distribution.rvs(self.n)
        x = torch.tensor(x, dtype=torch.float)

        # compute outputs using UM-Bridge
        model = umbridge.HTTPModel("http://localhost:49451", "forward")
        y = torch.zeros((self.n, 4))
        for i in range(n):
            if i % (n // 10) == 9:
                print(f"UM-Bridge Model Evaluations: {i + 1} / {n}")
            model_input = [x[i].tolist()]  # model input is a list of lists
            output = model(model_input)[0]
            y[i] = torch.tensor(output)

        # convert data to tensors and normalize to [0, 1]
        self.input_normalizer = sml.RangeNormalizer(x, dim=0)
        self.x = self.input_normalizer(x)
        self.output_normalizer = sml.RangeNormalizer(y, dim=0)
        self.y = self.output_normalizer(y)

    def __len__(self):
        return self.n

    def __getitem__(self, item):
        return self.x[item], self.y[item]


# define the dataset
n = 100
dataset = AerodynamicDataset(n)
train_dataset, test_dataset = torch.utils.data.random_split(dataset, [0.5, 0.5])
train_dataloader = torch.utils.data.DataLoader(
    train_dataset, batch_size=10, shuffle=True
)
test_dataloader = torch.utils.data.DataLoader(test_dataset)
```

```
UM-Bridge Model Evaluations: 10 / 100
UM-Bridge Model Evaluations: 20 / 100
UM-Bridge Model Evaluations: 30 / 100
UM-Bridge Model Evaluations: 40 / 100
UM-Bridge Model Evaluations: 50 / 100
UM-Bridge Model Evaluations: 60 / 100
UM-Bridge Model Evaluations: 70 / 100
UM-Bridge Model Evaluations: 80 / 100
UM-Bridge Model Evaluations: 90 / 100
UM-Bridge Model Evaluations: 100 / 100
```

1.3 2. Define and train the neural network

We build a deterministic neural network with a single hidden layer (also known as a shallow network) to approximate the mapping from the inputs to the outputs. The neural network architecture is defined using Torch and the training is handled by UQpy.

[3]:
```
# define the model
in_features = 5
width = 16
out_features = 4
network = nn.Sequential(
    nn.Linear(in_features, width),
    nn.ReLU(),
    nn.Linear(width, width),
    nn.ReLU(),
    nn.Linear(width, out_features),
)
model = sml.FeedForwardNeuralNetwork(network)

# define and run the training scheme
optimizer = torch.optim.Adam(model.parameters(), lr=1e-3)
scheduler = torch.optim.lr_scheduler.ReduceLROnPlateau(optimizer, patience=100)
trainer = sml.Trainer(model, optimizer, scheduler=scheduler)
trainer.run(train_data=train_dataloader, test_data=test_dataloader,␣
 ↪epochs=1_000)
```

The model is fully trained! We use the trained model to predict the outputs on our testing data, and rescale these outputs to their original ranges.

[4]:
```
# compute deterministic NN predictions
model.eval()
x_test = test_dataset.dataset.x[test_dataset.indices]
y_test = test_dataset.dataset.y[test_dataset.indices]
nn_prediction = model(x_test)
nn_prediction = nn_prediction.detach()

# rescale data and predictions
```

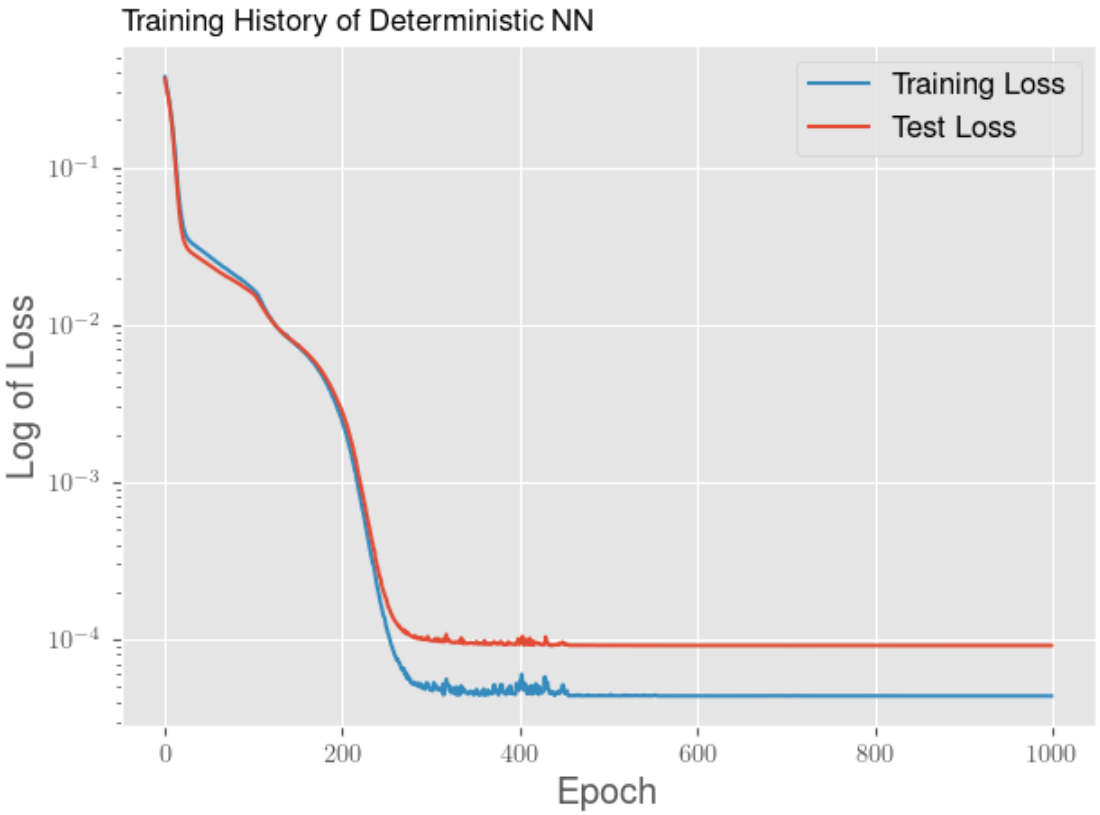
```
dataset.input_normalizer.decode()
dataset.output_normalizer.decode()
x_test = dataset.input_normalizer(x_test)
y_test = dataset.output_normalizer(y_test)
nn_prediction = dataset.output_normalizer(nn_prediction)
```

1.4 3. Plot the results

The hard work is done! The rest of this notebook plots the neural network predictions and compares them to the values from XFoil.

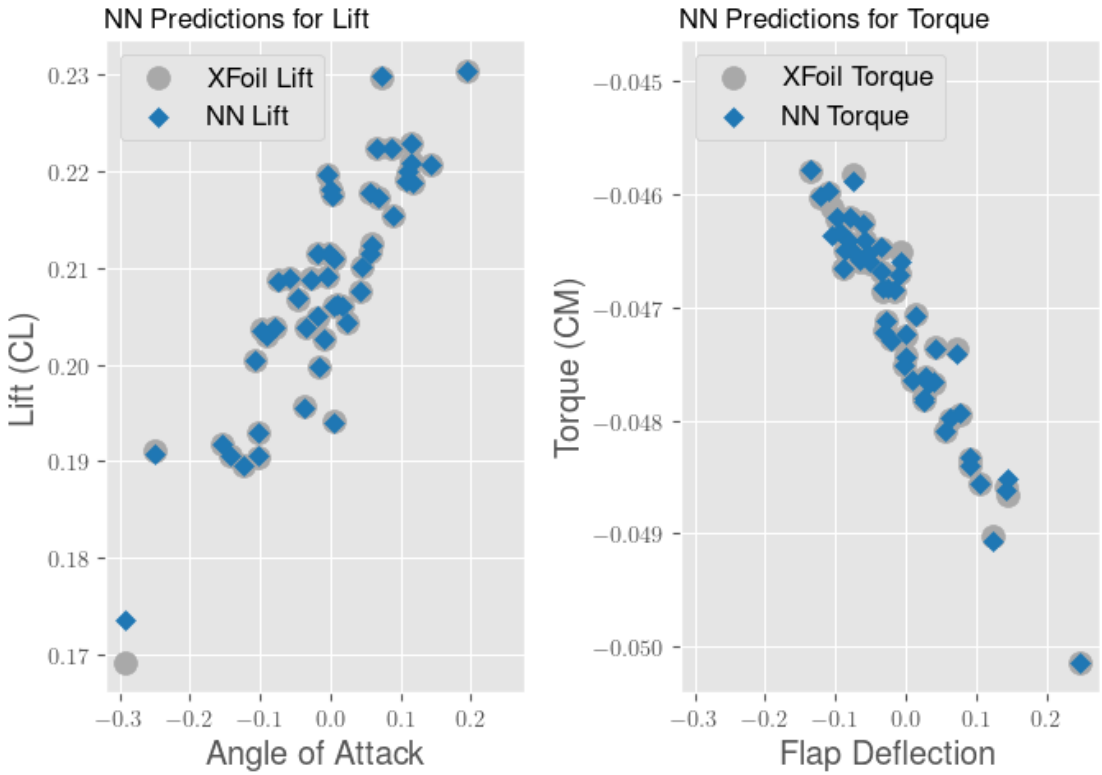
[5]:
```
import matplotlib.pyplot as plt
plt.style.use(["ggplot", "surg.mplstyle"])
```

[6]:
```
# plot training and testing loss
fig, ax = plt.subplots()
ax.semilogy(trainer.history["train_loss"], label="Training Loss")
ax.semilogy(trainer.history["test_loss"], label="Test Loss")
ax.set_title("Training History of Deterministic NN", loc="left")
ax.set(xlabel="Epoch", ylabel="Log of Loss")
ax.legend()
fig.tight_layout()
```



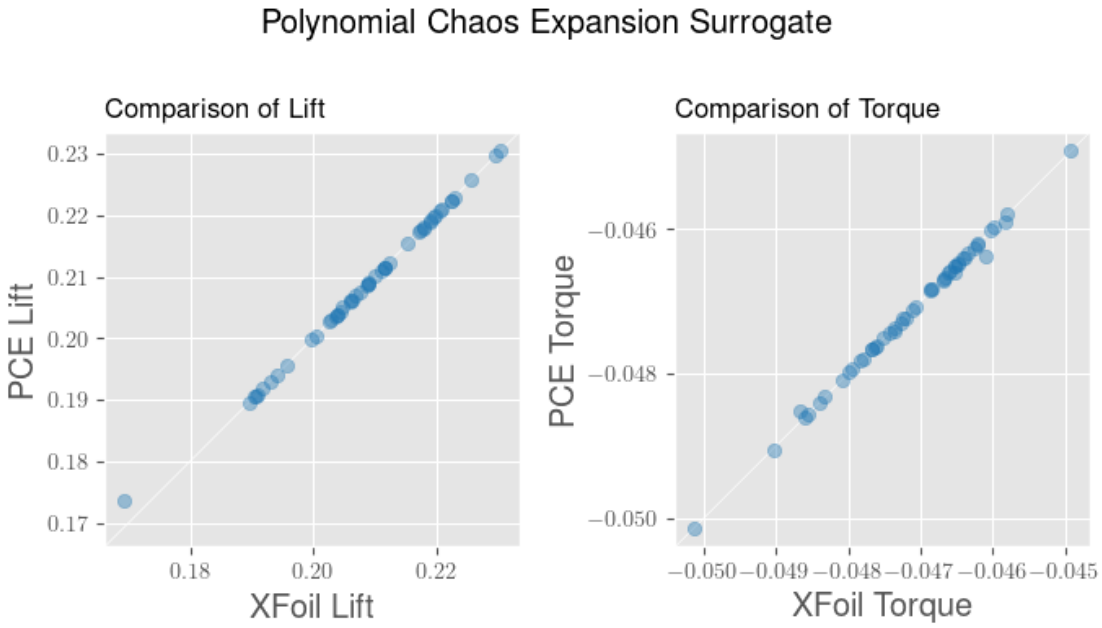
[7]:
```
# plot neural network and XFOIL results
fig, (ax0, ax1) = plt.subplots(figsize=(7, 5), ncols=2, sharex=True)
ax0.scatter(
    x_test[:, 0],
    y_test[:, 0],
    label="XFoil Lift",
    color="darkgray",
    marker="o",
    s=10**2,
)
ax0.scatter(
    x_test[:, 0],
    nn_prediction[:, 0],
    label="NN Lift",
    color="tab:blue",
    marker="D",
    s=6 ** 2,
)
ax0.set_title("NN Predictions for Lift")
ax0.set(xlabel="Angle of Attack", ylabel="Lift (CL)")
ax0.legend(loc="upper left", framealpha=1)

ax1.scatter(
    x_test[:, 4],
    y_test[:, 3],
    label="XFoil Torque",
    color="darkgray",
    marker="o",
    s=10**2,
)
ax1.scatter(
    x_test[:, 4],
    nn_prediction[:, 3],
    label="NN Torque",
    color="tab:blue",
    marker="D",
    s=6 ** 2,
)
ax1.set_title("NN Predictions for Torque")
ax1.set(xlabel="Flap Deflection", ylabel="Torque (CM)")
ax1.legend(loc="upper left", framealpha=1)
fig.tight_layout()
```



[8]:
```
# directly compare the XFoil and surrogate results
fig, (ax0, ax1) = plt.subplots(ncols=2, figsize=(7, 4))
ax0.scatter(y_test[:, 0], nn_prediction[:, 0], color="tab:blue", alpha=0.4,␣
 ↪zorder=2)
ax1.scatter(y_test[:, 3], nn_prediction[:, 3], color="tab:blue", alpha=0.4,␣
 ↪zorder=2)

# format the plots
ax0.set_title("Comparison of Lift")
ax0.set(xlabel="XFoil Lift", ylabel="PCE Lift")
ax1.set_title("Comparison of Torque")
ax1.set(xlabel="XFoil Torque", ylabel="PCE Torque")
ax1.set_yticks(ax1.get_xticks())
for ax in (ax0, ax1):
    xlim = ax.get_xlim()
    ax.plot(xlim, xlim, color="white", linewidth=0.5, zorder=1)
    ax.set_aspect("equal")
    ax.set(xlim=xlim, ylim=xlim)
fig.suptitle("Polynomial Chaos Expansion Surrogate")
fig.tight_layout()
```


Polynomial Chaos Expansion Surrogate
Comparison of Lift
PCE Lift
XFoil Lift
0.23
0.22
0.21
0.20
0.19
0.18
0.17
0.18
0.20
0.22
Comparison of Torque
PCE Torque
XFoil Torque
−0.046
−0.048
−0.050
−0.050 −0.049 −0.048 −0.047 −0.046 −0.045

[ ]: