train_bayesian_nn

September 24, 2024

1 UQpy at FrontUQ 2024

1.1 Deterministic Neural Network

This notebook trains a Bayesian 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 Bayesian neural network
3. Compute a distribution of predictions from our Bayesian neural network
4. Plot the results

Note this notebook is nearly identical to `train_deterministic_nn.ipynb` with minor modifications to define and train a Bayesian neural network

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. Generate training and testing data

Identical to the deterministic case, we use PyTorch's Dataset and DataLoader classes to define the training and testing data for the neural network.

This class is identical to the `AerodynamicDataset` in `train_deterministic_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 a Bayesian neural network

Here is where we deviate from the deterministic case. Bayesian neural networks learn a *distribution* of weights and biases rather than fixed values. They are useful as surrogates because, like Gaussian Process Regressions, they allow us to compute a distribution of outputs rather than a single point.

We use a combination of Torch and UQpy to define the Bayesian neural network. We put Torch's `ReLU` activation functions in betwen UQpy's `BayesianLinear` layers to construct our network. Both objects share the Torch Module as a parent class, so they can be placed side by side.

Bayesian neural networks are still data-driven models and are trained using (input, output) pairs using a gradient descent algorithm. The details of the algorithm differ from their deterministic counterparts and UQpy provides the `BBBTrainer` to implement a version of the Bayes-by-Backprop algorithm.

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

# define and run optimization
optimizer = torch.optim.Adam(model.parameters(), lr=1e-3)
scheduler = torch.optim.lr_scheduler.ReduceLROnPlateau(optimizer, patience=100)
bbb_trainer = sml.BBBTrainer(model, optimizer, scheduler=scheduler)
bbb_trainer.run(
    train_data=train_dataloader,
    test_data=test_dataloader,
    epochs=1_000,
    num_samples=10,
    beta=1e-6,
)
```

That completes the training of the neural network. Below we compute the outputs of the trained model by using the mean of its weights and by sampling from the weight's distributions. We can control the behavior of the weights (either sampling from the distributions or using their mean) using the `model.sample()` method.

[4]:
```
x_test = test_dataset.dataset.x[test_dataset.indices]
y_test = test_dataset.dataset.y[test_dataset.indices]

# compute mean predictions
model.sample(False)
with torch.no_grad():
    bnn_predictions = model(x_test)
bnn_predictions = bnn_predictions.detach()

# compute probabilistic predictions
model.sample(True)
n_samples = 1_000
samples = torch.empty((n_samples, *y_test.shape))
for i in range(n_samples):
    with torch.no_grad():
        samples[i] = model(x_test).detach()
bnn_std = torch.std(samples, dim=0)

# 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)
bnn_predictions = dataset.output_normalizer(bnn_predictions)
samples = dataset.output_normalizer(samples)
```

1.4 4. Plot results

That's it! We generated our training data, defined and trained a model, then computed its output. The rest of this notebook visualizes the results.

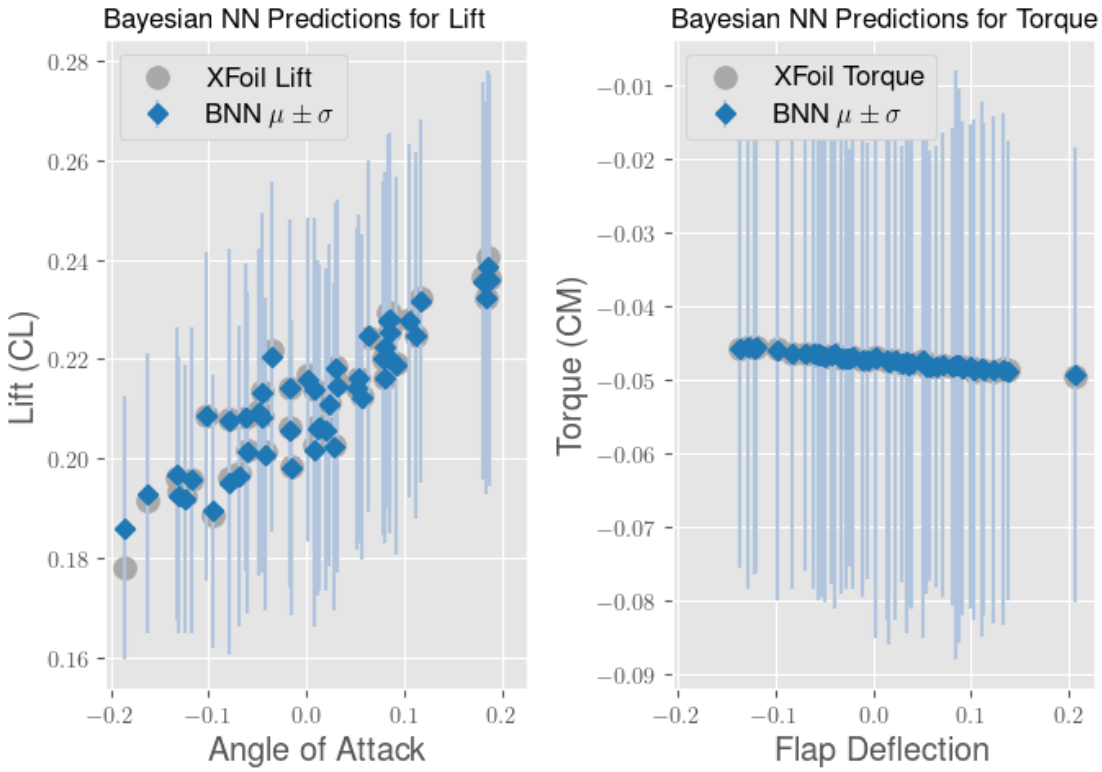
[5]:
```
import matplotlib.pyplot as plt
plt.style.use(["ggplot", "surg.mplstyle"])

# plot training history
fig, ax = plt.subplots()
ax.semilogy(bbb_trainer.history["train_loss"], label="Training Loss")
ax.semilogy(bbb_trainer.history["test_nll"], label="Test Loss")
ax.set_title("Training History", loc="left")
ax.set(xlabel="Epoch")
ax.legend()
fig.tight_layout()
```

[6]:
```
# plot predictions 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.errorbar(
    x_test[:, 0],
    bnn_predictions[:, 0],
    yerr=bnn_std[:, 0],
    label="BNN $\mu \pm \sigma$",
    color="tab:blue",
    ecolor="lightsteelblue",
    marker="D",
    linestyle="none",
)
ax0.set_title("Bayesian NN Predictions for Lift")
```

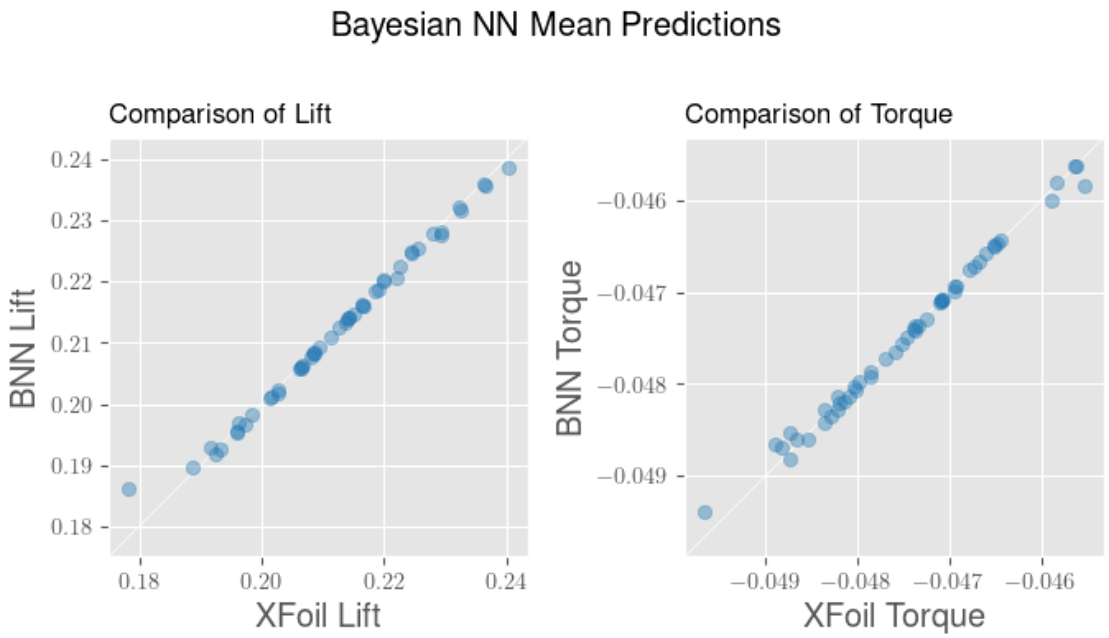
```
ax0.set(xlabel="Angle of Attack", ylabel="Lift (CL)")
ax0.legend(loc="upper left", framealpha=1)

ax1.scatter(
    x_test[:, 4],
    y_test[:, 3],
    label="XFoil Torque",
    color="darkgray",
    marker="o",
    s=10**2,
)
ax1.errorbar(
    x_test[:, 4],
    bnn_predictions[:, 3],
    yerr=bnn_std[:, 3],
    label="BNN $\mu \pm \sigma$",
    color="tab:blue",
    ecolor="lightsteelblue",
    marker="D",
    linestyle="none",
)
ax1.set_title("Bayesian NN Predictions for Torque")
ax1.set(xlabel="Flap Deflection", ylabel="Torque (CM)")
ax1.legend(loc="upper left", framealpha=1)
fig.tight_layout()
```



[7]:
```
fig, (ax0, ax1) = plt.subplots(ncols=2, figsize=(7, 4))
ax0.scatter(y_test[:, 0], bnn_predictions[:, 0], color="tab:blue", alpha=0.4,␣
 ↪zorder=2)
ax1.scatter(y_test[:, 3], bnn_predictions[:, 3], color="tab:blue", alpha=0.4,␣
 ↪zorder=2)

# format the plots
ax0.set_title("Comparison of Lift")
ax0.set(xlabel="XFoil Lift", ylabel="BNN Lift")
ax1.set_title("Comparison of Torque")
ax1.set(xlabel="XFoil Torque", ylabel="BNN 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("Bayesian NN Mean Predictions")
fig.tight_layout()
```


Bayesian NN Mean Predictions
Comparison of Lift
BNN Lift
XFoil Lift
0.24
0.23
0.22
0.21
0.20
0.19
0.18
0.18
0.20
0.22
0.24
Comparison of Torque
BNN Torque
XFoil Torque
−0.046
−0.047
−0.048
−0.049
−0.049
−0.048
−0.047
−0.046

[ ]: