This project has received funding from the European Union's Horizon Europe Programme

RESEARCH PROJECT EASA.2022.C25

D-3.3 CASE STUDY TRAINING MATERIAL

MODEL-SI

Digital Transformation - Case Studies for Aviation Safety Standards - Modelling and Simulation

An Agency of the European Union

Disclaimer

Funded by the European Union. Views and opinions expressed are however those of the author(s) only and do not necessarily reflect those of the European Union or the European Union Aviation Safety Agency (EASA). Neither the European Union nor EASA can be held responsible for them.

This deliverable has been carried out for EASA by an external organisation and expresses the opinion of the organisation undertaking this deliverable. It is provided for information purposes. Consequently it should not be relied upon as a statement, as any form of warranty, representation, undertaking, contractual, or other commitment binding in law upon the EASA.

DELIVERABLE NUMBER AND TITLE: MODEL-SI – D-3.3 Case study training material
CONTRACT NUMBER: EASA.2022.C25
CONTRACTOR / AUTHOR: ZHAW
IPR OWNER: European Union Aviation Safety Agency
DISTRIBUTION: Public

DATE: 12 November 2024

SUMMARY

This report is a deliverable document labelled D-3.3 about the “Training materials case study” of the research project number EASA.2022.C25 named MODEL-SI (Digital Transformation - Case Studies for Aviation Safety Standards - Modelling and Simulation).

The case study presented in deliverable D-2.1 [1] explores the potential of AI and ML to support electrical Vertical Take-Off and Landing (eVTOL) vehicles design and development. It advocates for changes in certification and aviation standards to accommodate these new methods. The case study presents the development of a data-driven Digital Twin (DT) using a combination of physics-based modelling, high-fidelity simulations, flight test data, and Machine Learning (ML) techniques. The DT could predict the entire flight envelope and load distributions. A probabilistic approach using Bayesian Neural Networks (BNN) is employed to assess the DT's trustworthiness and provide epistemic uncertainties for informed decision-making.

The physics-based models are comprehensively described in deliverable D-2.1 and are considered state-of-the-art in the aerospace industry. This report focuses on the innovative contributions of the MODEL-SI project, specifically the implementation and application of the ML methods. These approaches were used to build a data-driven model, which was then implemented within the comprehensive flight simulation model (FSM). In particular, it delves into the multi-fidelity Bayesian Neural Networks (BNN) library, a tool designed for training, testing, and deploying BNNs that can handle data of varying fidelity levels. The library is particularly suited for scenarios where multiple levels of data are available and the goal is to create accurate models with associated uncertainty estimates.

CONTENTS

ABBREVIATIONS

ACRONYM	DESCRIPTION
AI	Artificial Intelligence
ANN	Artificial Neural Networks
BNN	Bayesian Neural Networks
CFD	Computational Fluid Dynamics
DF	Data Fusion
DNN	Deep Neural Network
DT	Digital Twin
EASA	European Union Aviation Safety Agency
eVTOL	Electrical Vertical Take-Off and Landing
FEM	Finite Element Method
FSM	Flight Simulation Model
FT	Flight Test
HC	Helicopter
HF	High-Fidelity
GP	Gaussian process
LF	Low-Fidelity
LHS	Latin Hypercube Sampling
MF	Mid-Fidelity
ML	Machine Learning
NN	Neural Networks
NS	Navier-Stokes
TL	Transfer Learning
VLM	Vortex Lattice Method
VTOL	Vertical Take-Off and Landing
UAV	Unmanned Aerial Vehicle
UQ	Uncertainty Quantification

1. Introduction

The case study of the MODEL-SI project delved into the development of physics-based and data-driven methodologies to build our comprehensive FSM. This project explores a range of methodologies to develop accurate and practical flight load envelopes for our multi-rotor eVTOL aircraft. The investigation focuses on modelling a broad spectrum of potential configurations, control strategies, flight conditions, and variations in rotor speed and acceleration, accounting for both normal operations and failure scenarios. The case study is

Our proposed methodology begins with low-fidelity (LF) physics-based models, supplemented by a limited set of mid-fidelity (MF) and high-fidelity (HF) data points, to create and progressively refine a data-driven model. It can potentially be used throughout the development, certification, and operational phases of the eVTOL. This iterative process integrates numerical simulations and experimental data as they become available. Importantly, there is no static final model; the digital twin (DT) is designed to adapt to aircraft customizations and reconfigurations, maintaining its relevance over the vehicle's entire lifecycle.

The data-driven model is integrated into our FSM, which exploits a modular design strategy that divides the system into independent, self-contained modules, each with distinct functions and interfaces. This approach improves flexibility, scalability, and reusability, making it easier to manage and adapt complex systems. Modular designs are especially valuable in multidisciplinary environments. Figure 1 illustrates the comprehensive modeling strategy.

The Flight Mechanics module (nonlinear equilibrium/trim) is represented on the left, while the Aeroelastic module (dynamic stability) is on the right. The Flight Mechanics module is designed for an in-depth analysis of the eVTOL's rigid-body flight mechanics and dynamics. In contrast, the Aeroelastic module focuses on static deflections, linear stability, stability margins, and dynamic response, which can be analyzed using either the linearized system (in time or frequency domains) or the full nonlinear system through time integration.

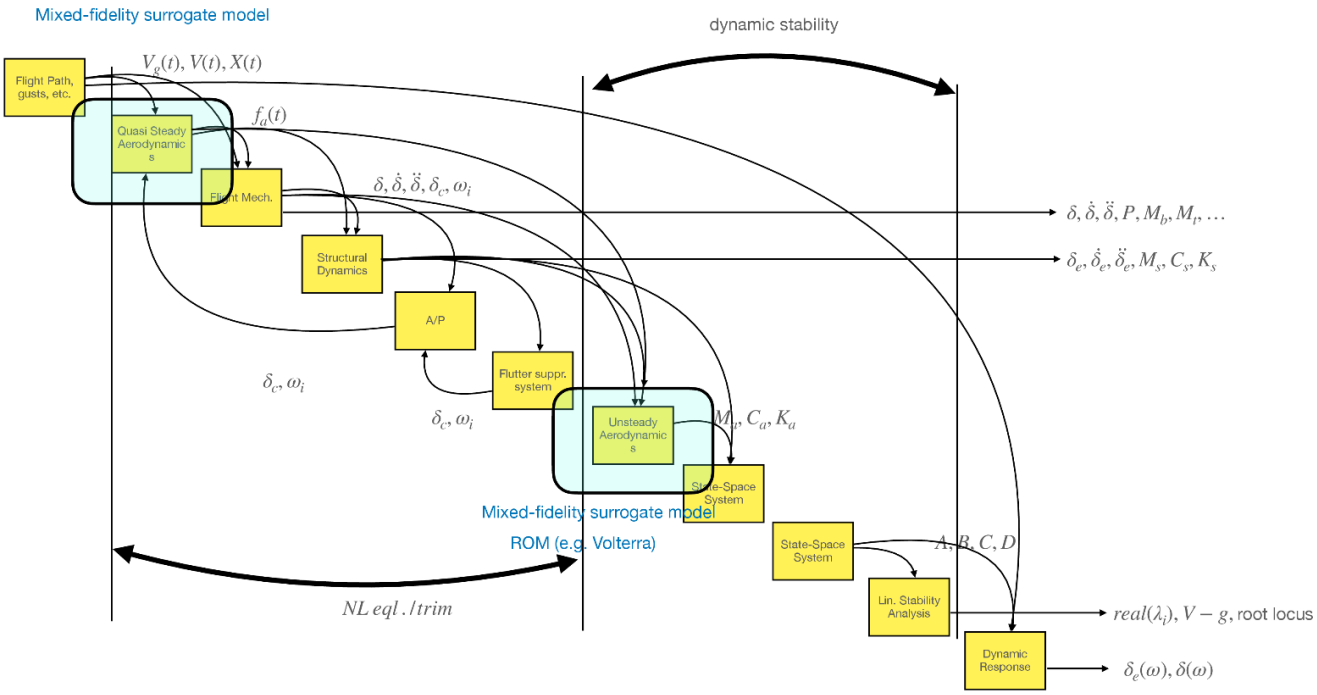

Figure 1: Comprehensive modelling strategy.

The blue boxes indicate the models that were replaced by the ML/AI contribution To build those models, the Multi-Fidelity BNN (MF-Baynet) library is designed to support the training, testing, and deployment of Bayesian

Neural Networks (BNNs) with different fidelity levels in a data fusion context. This library is particularly suited for scenarios where multiple levels of data fidelity are available, and the goal is to create models that can leverage this multi-fidelity data to make robust predictions with uncertainty estimates. This library is based on methodologies and approaches discussed in the paper Vaiuso. et al. [2]. This method has been shown to outperform traditional approaches like Co-Kriging in both accuracy and uncertainty quantification. The approach is particularly valuable in aerospace engineering for tasks such as predicting transonic aerodynamic loads where different fidelity levels (e.g., low-fidelity panel methods, mid-fidelity RANS simulations, and high-fidelity CFD) are available.

2. Requirements

2.1 Functional Requirements

Data Handling and Preprocessing: The library must support loading datasets from various formats and perform preprocessing steps such as normalization, augmentation, and feature selection. The system must provide the ability to split data into training, validation, and test sets with configurable ratios.
Model Training: The library must allow for training Bayesian Neural Networks (BNNs) across different fidelity levels. It must support Transfer Learning (TL) to fine-tune models from one fidelity level using data from another.
Prediction and Uncertainty Quantification: The system must enable predictions with uncertainty estimates, allowing for multiple stochastic forward passes to quantify predictive uncertainty. The library must allow for predictions using both trained BNN models or other custom models, integrating the error results from various fidelities.
Model Testing and Evaluation: The library must provide functions to evaluate trained models on test datasets, outputting metrics such as error rates and uncertainty measures. It must support the comparison of different models trained on varying fidelity data.
Model Deployment: The library must support deploying trained models as a service, accessible via a server to perform real-time predictions on incoming data. The system must provide configuration settings for server deployment, such as host address, port, and device selection (CPU/GPU).
Result Visualization: The library must provide utilities for visualizing results, including correlation matrices and prediction vs. actual output plots. It must allow saving and exporting these visualizations for further analysis.

2.2 Non-Functional Requirements

Performance: The library should be optimized to handle large datasets and complex model architectures efficiently, minimizing training and prediction time, especially when using GPU acceleration.
Scalability: The system should be scalable, allowing users to easily extend it with new model types or additional data preprocessing and augmentation techniques.
Usability: The library should have a clear and intuitive API, with comprehensive documentation, including examples, to facilitate easy integration into existing workflows. Configuration files should be user-friendly and well-documented to allow for easy customization of model training, testing, and deployment parameters.
Robustness: The library should handle edge cases gracefully, such as missing or malformed data, and provide meaningful error messages to aid in debugging. It should include mechanisms for checking and validating input data and model configurations to prevent common errors during runtime.
Portability: The system should be platform-independent, capable of running on various operating systems, and should not rely on any specific proprietary software.
Security: When deployed as a server, the library must ensure that the service is secure, with protections against unauthorized access or data breaches. Sensitive data handling should be implemented with care, ensuring that no personal or proprietary data is exposed during the model training or prediction processes.
Maintainability: The codebase should be well-structured and modular, allowing for easy maintenance, updates, and extensions by future developers. The system should include automated tests for key functionalities to ensure that changes do not introduce bugs.

3. Use Cases

The use cases outlined below illustrate how users can leverage this library for executing the main tasks. These scenarios provide a comprehensive overview of the library's capabilities.

1. **Model Training with Multi-Fidelity Data**
   Use Case: Train a MF-BNN model using datasets of varying fidelity levels.
   Description: Users load multi-fidelity datasets, configure model settings, and train BNN models to learn from different fidelities data. Transfer Learning can be employed to fine-tune models trained on LF data using MF data, improving prediction accuracy and uncertainty quantification.
2. **Data Augmentation**
   Use Case: Augment datasets to improve model robustness.
   Description: Users generate additional data points by adding noise, interpolating between existing points, or using model-based predictions to create new examples.
3. **Prediction with Uncertainty Estimation**
   Use Case: Make predictions using a trained BNN model, with uncertainty quantification.
   Description: Users input new data into the trained BNN model to generate predictions along with associated uncertainty estimates. This is crucial for applications where understanding confidence in predictions is important, such as in safety-critical systems.
4. **Model Evaluation**
   Use Case: Evaluate the performance of trained models on a test dataset.
   Description: Users test trained BNN models on unseen data to assess their predictive performance. Error metrics and uncertainty measures are calculated, helping users understand the model's accuracy and reliability across different fidelity levels.
5. **Model Deployment**
   Use Case: Deploy trained BNN models as a web service.
   Description: Users deploy the BNN models on a server to make them accessible via API calls. This allows other applications or users to send input data to the server and receive predictions in real-time, enabling the integration of BNN models into larger systems.

4. Project Structure

4.1 Source root structure

The next schema represents the source structure with all the codebase files and folders. Python executable codes implement all the use cases.

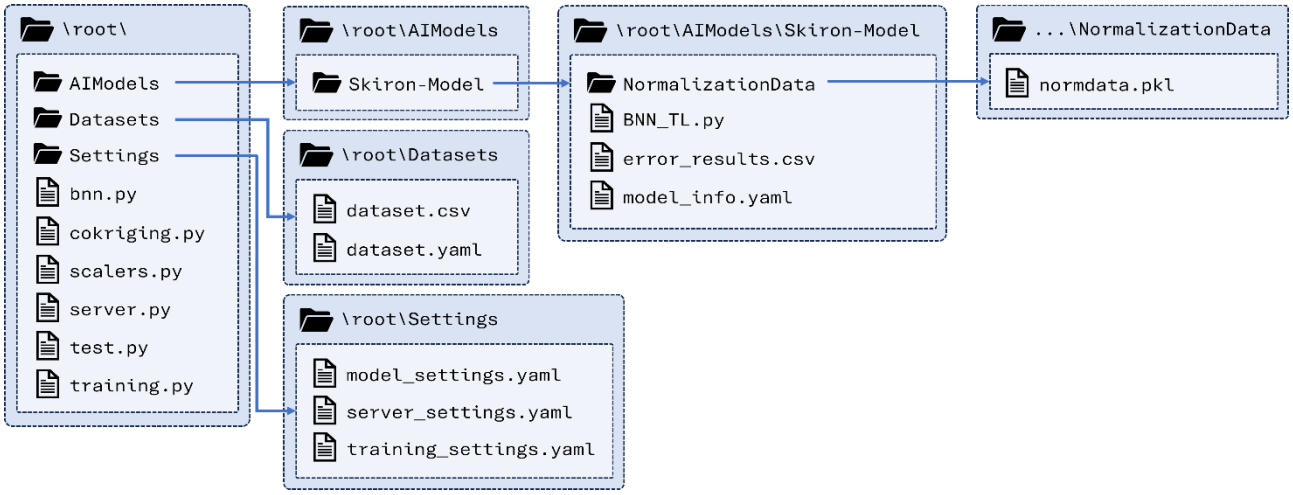

Figure 2: Quick and dirty' pathway.

4.1.1 MVC Schema for data structures

The Multi-Fidelity BNN Library can be conceptualized within the Model-View-Controller (MVC) architectural pattern as follows:

4.1.1.1 Models

- **ML Model (bnn.py->BNN class, cockriging.py->CK class):** The core Bayesian Neural Network (BNN) and CoKriging (CK) models that perform the primary Data Fusion computations. They encapsulate the logic for training, data fusion, prediction, model testing and comparison and uncertainty quantification using multi-fidelity data.
- **Dataset Handling (bnn.py->BNNDataset class):** Manages the dataset, including loading, splitting, and preprocessing, ensuring the data is ready for training and evaluation.

4.1.1.2 Views

- **Visualization Tools (training.py, utils.py):** Includes utilities for plotting and analyzing model predictions, correlations, and performance metrics. These provide the necessary feedback to users, helping them interpret the model's behavior and outcomes.

4.1.1.3 Controllers

- **Training Controller (training.py):** Orchestrates the model training process by loading configurations, managing data flow, training models, and saving outputs. It handles the interaction between the model and the dataset.
- **Server Controller (server.py):** Manages the deployment of models as web services, handling incoming prediction requests and returning model outputs. It acts as the interface between the model and external applications.

4.2 Source structure definitions

Here, the directory and the single files porpoises are explained (refer to Figure 2).

4.2.1 AI Models

This directory contains the output models generated by the training process, following the settings defined.

4.2.2 Datasets

This directory contains the definitions and settings of the datasets used for the training.

- **dataset.yaml:** Contains the parameters and configurations of the dataset for training.

4.2.3 Settings

This directory contains YAML configuration files that define various settings used across the project.

- **model_settings.yaml:** Contains the hyperparameters and configurations for the Bayesian Neural Network models.
- **server_settings.yaml:** Defines the settings required to deploy the model on a server, including the host address, port number, and device information.
- **training_settings.yaml:** Specifies the training-related configurations, including training epochs, batch sizes, learning rates, and paths for saving models.

4.2.4 BNN.py

This module defines the Bayesian Neural Network (BNN) classes and methods for handling multi-fidelity data.

- **BNNDataset (class):** A custom dataset class that handles loading and managing multi-fidelity datasets. It includes methods for splitting the data into training, validation, and testing sets.
- **BNN (class):** The core Bayesian Neural Network class that handles the architecture, forward passes, training loops, and prediction routines. This class supports uncertainty estimation by performing multiple stochastic forward passes.
- **test_multiple_models (static function):** A utility function to test multiple BNN models on a given dataset, useful for comparing the performance of models trained on different fidelity levels.

4.2.5 Cokriging.py

This module defines the Co-Kriging (CK) model, which is used to combine predictions from low-fidelity and high-fidelity models.

- **CK (class):** The Co-Kriging class that integrates predictions from a low-fidelity BNN model with a Gaussian Process (GP) to improve the prediction accuracy for higher-fidelity data. This class includes methods for training the GP model, making predictions, and saving/loading the CK model.

4.2.6 Scalers.py

This module defines custom scalers used for normalizing data.

- **MinMaxScaler:** A custom implementation of the Min-Max scaler that scales the features to a specified range. This scaler is particularly useful in neural network training as it ensures that the input features are within a range that facilitates convergence.

4.2.7 Server.py

This module provides the functionality to deploy a trained BNN model on a server. It allows the model to be used as a service, accepting input data over a network, and returning predictions.

- **run_server (static function):** A function that sets up and runs the server using the settings defined in server_settings.yaml.
- **handle_request (static function):** A function to handle incoming requests to the server, process the input data, run the model, and return predictions.

4.2.8 Test.py

This script is responsible for testing the trained BNN and CK models on a test dataset. It performs the following tasks:

- **Model Loading:** Loads the trained models from their saved state.
- **Dataset Loading:** Loads the test dataset and applies the necessary normalization defined in model_settings.yaml.
- **Model Testing:** Runs the models on the test dataset, compares predictions against ground truth, and calculates error metrics.
- **Result Saving:** Saves the results of the model testing to a CSV file of the specified working folder defined in server_settings.yaml and model_settings.yaml.

4.2.9 Training.py

This script handles the end-to-end training process of the BNN models, including:

- **Settings Loading:** Reads all the configurations from the YAML files.
- **Data Preparation:** Loads and preprocesses the datasets, applies normalization, and splits them into training, validation, and testing sets.
- **Model Training:** Trains the low-fidelity, mid-fidelity, and fine-tuned models, as well as the Co-Kriging model.
- **Model Saving:** Saves the trained models and their configurations for later use.
- **Result Plotting:** Plots the predictions of the trained models and compares them with the validation data.

4.2.10 Utils.py

This utility module provides helper functions that are used across various parts of the project.

4.3 Settings Parameters

In this section each parameter of yaml configuration file is explained

4.3.1 model_settings.yaml

Contains the hyperparameters and configurations for the Bayesian Neural Network models.

- **MODEL_NAME:** Specifies the name of the Transfer Learning (TL) model, typically representing the final model integrating multiple fidelity levels.
- **MODEL_NAME_MF:** Name assigned to the mid-fidelity model.
- **MODEL_NAME_LF:** Name assigned to the low-fidelity model.
- **MODEL_NAME_CK:** Name assigned to the Co-Kriging model.
- **MU:** Mean of the prior distribution for the Bayesian layers in the low-fidelity/TL model, influencing the initial belief about the weights.
- **MU_MF:** Mean of the prior distribution for the Bayesian layers in the mid-fidelity model.
- **STD:** Standard deviation of the prior distribution for the Bayesian layers in the low-fidelity/TL model, determining the spread of initial weight distributions.
- **STD_MF:** Standard deviation of the prior distribution for the Bayesian layers in the mid-fidelity model.
- **UNITS:** Defines the architecture of the low-fidelity/TL model by specifying the number of units (neurons) in each hidden layer.
- **UNITS_MF:** Specifies the architecture of the mid-fidelity model by defining the number of units in each hidden layer.
- **DEVICE:** The computational device used for training and inference, either "cpu" or "cuda" (GPU).
- **SEED:** The random seed used to ensure reproducibility of results during training (warning: remember that the BNN model is, by definition, non-deterministic).

4.3.2 server_settings.yaml

Defines the settings required to deploy the model on a server, including the host address, port number, and device information.

- **MODEL_PATH**: Path to the saved model file that the server will load and use for making predictions.
- **NORM_DATA_PATH:** Path to the normalization data used during model training, required to preprocess input data consistently.
- **DEVICE:** Specifies the device on which the model will run, either "cpu" or "cuda".
- **PORT_NO:** The port number on which the server will listen for incoming connections.
- **HOST_ADDRESS:** The host address for the server; "0.0.0.0" indicates that the server will listen on all available network interfaces.

4.3.3 training_settings.yaml

Specifies the training-related configurations, including training epochs, batch sizes, learning rates, and paths for saving models.

- **MODEL_PATH**: Directory where the trained models will be saved.
- **DATASET_YAML_PATH:** Path to the YAML file containing dataset configurations.
- **PATIENCE_LF:** The number of epochs with no improvement in the validation loss after which training of the low-fidelity model will stop.
- **N_EPOCHS_LF:** Maximum number of epochs to train the low-fidelity model.
- **BATCH_SIZE_LF:** The number of samples per gradient update for the low-fidelity model.
- **LR_LF:** Learning rate for optimizing the low-fidelity model.
- **PATIENCE_MF:** Early stopping patience for mid-fidelity model training.

- **N_EPOCHS_MF:** Maximum number of epochs to train the mid-fidelity model.
- **BATCH_SIZE_MF:** The number of samples per gradient update for the mid-fidelity model.
- **LR_MF:** Learning rate for optimizing the mid-fidelity model.
- **PATIENCE_TL:** Early stopping patience for transfer learning model training.
- **N_EPOCHS_TL:** Maximum number of epochs to train the transfer learning model.
- **BATCH_SIZE_TL:** The number of samples per gradient update for the transfer learning model.
- **LR_TL:** Learning rate for optimizing the transfer learning model.
- **N_LAYER_TO_UNFREEZE:** Number of layers to unfreeze during fine-tuning in transfer learning. Determines the depth of the fine-tuning process.
- **TRAIN_LF:** Boolean flag indicating whether to train the low-fidelity model.
- **TRAIN_MF:** Boolean flag indicating whether to train the mid-fidelity model.
- **TRAIN_TL:** Boolean flag indicating whether to perform transfer learning.
- **TRAIN_CK:** Boolean flag indicating whether to train the Co-Kriging model.

4.3.4 Datasets/dataset_settings.yaml

Specifies the configuration of training dataset, including csv file location, description, and labels, as well as splitting proportions.

- **DATASET_LOCATION:** Path to the dataset file used for training and evaluation.
- **SEP:** The delimiter used in the dataset file (e.g., semicolon, comma).
- **DATASET_DESCRIPTION:** A brief description of the dataset, including any augmentations or noise added.
- **INPUT_LABELS:** List of input features/variables used in the model, such as angles of attack, sideslip, and RPM values.
- **OUTPUT_LABELS:** List of output variables that the model predicts, such as thrust and torque values for different propellers.
- **FIDELITY_COLUMN_NAME:** The name of the column that identifies the fidelity level of each data point in the dataset.
- **FIDELITIES:** List of fidelity levels included in the dataset, typically low-fidelity (LF) and mid-fidelity (MF).
- **LF_TRAIN_SIZE:** Proportion of the low-fidelity data used for training the model.
- **MF_TRAIN_SIZE:** Proportion of the mid-fidelity data used for training the model.
- **MF_VALID_SIZE:** Proportion of the mid-fidelity data used for validating the model during training.

5. Implementing and serving a model

5.1 Introduction

This section provides a step-by-step guide on how to effectively use the library to implement and serve a machine learning MF-Baynet model. You'll learn how to configure your model, train it on multi-fidelity data, evaluate its performance, and deploy it as a web service for real-time predictions. This example is designed to help you integrate the library into your workflow, ensuring a smooth transition from development to production.

5.2 Configure the settings

Before running the training or testing scripts, ensure that the YAML files in the Settings/ directory are correctly configured. These files should define all necessary hyperparameters, file paths, and other configurations needed by the models and scripts.

5.3 Train a model

Run the training.py script to start the training process. This script will:

- Load the dataset.
- Normalize the data.
- Train the low-fidelity, mid-fidelity, and transfer learning models.
- Train the Co-Kriging model using the low-fidelity model as a basis.
- Save all trained models and their configurations.

The results of model training (model file, training curves and model comparisons) will be saved in the model path defined in the model_settings.
Run the file by typing in the bash the following code:

```
python training.py
```

5.4 Test a model

After training, use the test.py script to evaluate the models on a test dataset. This script will:

- Load the trained models and test dataset.
- Normalize the test data using the same scaler used during training.
- Perform predictions using the trained models.
- Save the prediction results and error metrics for further analysis.

The results of model testing (model comparisons and output plots) will be saved in the model path defined in the model_settings.
Run the file by typing in the bash the following code:

```
python test.py
```

5.5 Deploy the model

Once the model is trained and validated, it can be deployed using the server.py script. This script sets up a server that can accept input data, run the model, and return predictions. The server can be configured using the settings in server_settings.yaml.
Run the file by typing in the bash the following code:

```
python server.py
```

6. Generate a custom model

6.1 Example

In this section the procedure for writing a custom Python training script will be explained using the functions defined on training.py.

1. Import necessary functions from library (it's supposed that the new script is created into the root directory):

```
from training import import_settings, setup_model_directory,
load_and_normalize_datasets, save_model_data, test_models
from bnn import BNNDataset, BNN
import copy
```

2. Import yaml settings variable:

```
model_settings, training_settings, dataset_settings = import_settings()
```

3. Setup model PATH directory. The function returns a Boolean variable that indicates if the selected path already exists or not

```
path_exists = setup_model_directory(training_settings["MODEL_PATH"])
```

4. Load the dataset normalized by using the settings defined in the yaml file. The return value "normalized_datasets" is a list that contains the all the normalized datasets divided into different fidelities, following the fidelities definition on dataset settings yaml file. If the path already exists, a warning message will appear.

```
normalized_datasets, scaler = load_and_normalize_datasets(
     dataset_settings,
     path_exists,
     training_settings["MODEL_PATH"])
```

5. Load the datasets into classes using the BNNDataset constructor. Use the appropriate index value to refer to the desired fidelity.

```
dataset_fidelity_x = BNNDataset(
     normalized_datasets[index],
     model_settings["INPUT_LABELS"],
     model_settings["OUTPUT_LABELS"],
     device=model_settings["DEVICE"])
```

6. Split each dataset in training, validation, or training, validation, and test (It's recommended to get the test-set from the highest fidelity dataset)

```
train_ dataset_1, valid_ dataset_1 = dataset_fidelity_x.train_val_split(
     dataset_settings["LF_TRAIN_SIZE"],
     model_settings["SEED"])
```

```
train_ dataset_2, valid_ dataset_2, test_ dataset_2 =
     dataset_fidelity_x.train_val_test_split(
          dataset_settings["LF_TRAIN_SIZE"],
          dataset_settings["LF_VALID_SIZE"],
          model_settings["SEED"])
```

7. Train the base model using the lowest fidelity dataset. First, build the model using BNN class constructor.

```
bnn_lf_model = BNN(
     in_dim = len(model_settings["INPUT_LABELS"]),
     out_dim = len(model_settings["OUTPUT_LABELS"]),
     mu = model_settings["MU"],
     std = model_settings["STD"],
     units = model_settings["UNITS"],
     denseOut = False,
     dropout = False,
     device = model_settings["DEVICE"],
     activation = nn.LeakyReLU(),
     model_name = model_settings["MODEL_NAME_LF"])
```

8. Then, start the training process using the training function. Note: this function utilizes different parameters. Refer to the codebase documentation for further information on the use of each parameter.

```
bnn_lf_model.train(
     train_dataset_1,
     valid_dataset_1,
     patience = training_settings["PATIENCE_LF"],
     n_epochs = training_settings["N_EPOCHS_LF"],
     batch_size = training_settings["BATCH_SIZE_LF"],
     lr = training_settings["LR_LF"],
     history_plot_saving_path = training_settings["MODEL_PATH"],
     showPlotHistory = True)
```

9. Save the model weight in .pt format in the settings path

```
bnn_lf_model.save(f"{
training_settings["MODEL_PATH"]}{bnn_lf_model.model_name}.pt",
subfolder="")
```

10. To increasingly freeze part of the model to train the higher fidelity dataset, copy the base model, set gradients to false and unfreeze a few layers (the number of layers to freeze is a hyperparameter that should be optimized for each problem. It's a number between 1 and the number of the network layer already trained, minus one).

```
bnn_df_model = copy.deepcopy(bnn_lf_model)
bnn_df_model.model_name = model_settings["MODEL_NAME"]
bnn_df_model.setModelGradients(False)
all_layers = bnn_df_model.getAllLayersName()
bnn_df_model.setModelGradients(
     True,
     layers = all_layers[-training_settings["N_LAYER_TO_UNFREEZE"]:])
```

11. Train the model following the training settings

```
bnn_df_model.train(
     train_dataset_2,
     valid_dataset_2,
     n_epochs = training_settings["N_EPOCHS_TL"],
     lr = training_settings["LR_TL"],
     restoreBestModel = True,
     patience = training_settings["PATIENCE_TL"],
     batch_size = training_settings["BATCH_SIZE_TL"],
     history_plot_saving_path = MODEL_PATH,
     showPlotHistory = True)
```

12. Finally, save the model

```
bnn_df_model.save(f"{
training_settings["MODEL_PATH"]}{bnn_df_model.model_name}.pt",
subfolder="")
```

13. Dump training data

```
save_model_data(
     model_settings,
     dataset_settings,
     training_settings,
     training_settings["MODEL_PATH"],
     bnn_df_model= bnn_df_model)
```

14. Test models using test set and save error results.

```
test_models(
     models_to_test = [bnn_lf_model, bnn_df_model],
     test_dataset_2,
     scaler,
     model_settings["OUTPUT_LABELS"],
     training_settings ["MODEL_PATH"])
```

7. Model deployment on server

7.1 Example of model deployment

Once the model is saved, it can be loaded in a server environment for API distribution. The server is personalized to serve the prediction for a single use case (the default digital twin), but it can be configured to get personalized API. The prediction output is defined in the function pred_bnn inside server.py file.

7.1.1 Request format

In the default example, if the server is running on host 0.0.0.0 with port 7878, it's possible to send an HTTP POST request to the method "/predict"

```
http://127.0.0.1:7878/predict
```

The POST message body should contain a JSON object with input data, as the example below:

```
{
     "aoa": 12.5,
     "aos": 3.7,
     "u_inf": 45.8,
     "PP": 0,
     "FR": 2100,
     "FL": 2100,
     "RR": 2100,
     "RL": 2100,
     "attempt": 10
}
```

The mandatory headers for he POST message are:

```
Content-Type: application/json
```

7.1.2 Response format

The response format is represented by a JSON file with the desired output calculated by the MF-Baynet model defined in the server_settings. The model output will add the following prefixes "mean_" and "std_" for mean and std values for each desired output.

```
{'mean_T_PP': 0.5643825805187226, 'mean_Q_PP': 0.02320480750873685,
'mean_T_FR': 48.95215629577637, 'mean_Q_FR': -2.0664690113067627,
'mean_T_FL': 54.0648059463501, 'mean_Q_FL': 2.0810022830963133,
'mean_T_RR': 51.39777896881103, 'mean_Q_RR': 2.1484628462791444,
'mean_T_RL': 51.652060966491696, 'mean_Q_RL': -2.0470683252811432,
'std_T_PP': 0.09293364033901781, 'std_Q_PP': 0.005035503091609833,
'std_T_FR': 1.1895130950415154, 'std_Q_FR': 0.054163867586413776,
'std_T_FL': 0.8112419560985902, 'std_Q_FL': 0.031835998291469506,
'std_T_RR': 0.7763757549238378, 'std_Q_RR': 0.03192960623290125,
'std_T_RL': 0.690751796017016, 'std_Q_RL': 0.023788539976873916}
```

Bibliography

[1] Andrea Pedrioli, Marcello Righi. MODEL-SI, D-2.1 Case study report. Reserach Project EASA.2022.C25, 2023.

[2] Andrea Vaiuso, Gabriele Immordino, Marcello Righi, Andrea Da Ronch. Multi–Fidelity Bayesian Neural Network for Uncertainty Quantification in Transonic Aerodynamic Loads. In *ArXiv* 2407.05684, 2024.

European Union Aviation Safety Agency

Konrad-Adenauer-Ufer 3
50668 Cologne
Germany

Mail EASA.research@easa.europa.eu
Web www.easa.europa.eu

An Agency of the European Union