ML-TFLAT User Guide

Table of Contents

[Introduction 1](#_Toc65609878)

[Numerical Model (MATLAB) 1](#_Toc65609879)

[Setting Options 1](#_Toc65609880)

[Setting Data Generation Parameters 1](#_Toc65609881)

[Running the Model 1](#_Toc65609882)

[Data Output 1](#_Toc65609883)

[Preprocessing of Data 1](#_Toc65609884)

[Neural Network (Python) 2](#_Toc65609885)

[Prerequisites 2](#_Toc65609886)

[Setting Options 2](#_Toc65609887)

[Setting Neural Network Parameters 2](#_Toc65609888)

[Training a Neural Network 2](#_Toc65609889)

[Training Data Outputs 2](#_Toc65609890)

[Loading a Saved Neural Network 2](#_Toc65609891)

[Testing a Neural Network 2](#_Toc65609892)

# Introduction

The ML-TFLAT is composed of two components at this point in its development: a numerical model to simulate the fatigue damage of the current setup, and a neural network structure designed to build, train, and test neural networks using data produced from the numerical model.

## Required Libraries

Python has the advantage of installable libraries that provide well-documented and advanced functionality to users, but these libraries require that they be installed on a system that is running programs that utilize them. Several libraries are used in the Python-based sections of the ML-TFLAT, but these are all included in the Individual Edition of the Anaconda package located at <https://www.anaconda.com/>. It is highly recommended that this package be used to install the necessary libraries and run the Python-based components of the ML-TFLAT.

# Numerical Model (MATLAB)

The numerical model is built in MATLAB and only utilizes standard libraries for its operation, so there are no special requirements for operation. The program is designed to operate out of the folder that the *ModelShell.m* file is located in, which should be located in the *Numerical Model v4* file.

## Setting Options and Material Parameters

The model is supplied with 4 *Config* folders and four *ModelShell.m* files, each appended with a matching identification number. These are provided for pre-configured simultaneous data generation, and each shell is independently configured through their respective *Config* folder. The Settings and Material configuration is the same regardless of the shell being used. The file names and locations cannot be changed without modifying the program as the model pulls data from them based on the given names and their location related to the root *Numerical Model v4* folder.

### Adjusting Settings

To change the Settings for the model, open the *Settings.xls* file. The two settings available define the number of nodes and the cycle count for data export from the model. **Note:** The number of nodes is equal to the setting value plus one due to how the model operates.

Increasing the number of nodes increases the number of elements used in the model, which increases run time in exchange for higher accuracy. Increasing the cycle count for output changes the rate at which the data is exported from the model and can be adjusted to change the size of output files. It is recommended to keep the number lower though as it reduces the memory demand on the system.

### Adjusting Material Parameters

To change the Material Parameters for the model, open the *MaterialParameters.xls* file. The available parameters are used by the model in simulating the material and can be changed to represent other materials. Each entry is labeled with the parameter name, the type of material in square brackets, and the units of the material property in parentheses. To ensure proper model function, the properties should be kept in the same units as presented.

## Setting the Model Configuration

The model is designed to take a list of values from the *ModelConfigurations.xls* file for each of the operating parameters and then iterate through all combinations of the values. The available configuration parameters for the designed setup are, in order: Temp Out (°C), Temp In (°C), Outer Radius (m), Inner Radius (m), Pipe Length (m), Flow Pressure (MPa), and Flow Rate (m3/s).

While the model is capable of iterating continuously through the configurations, it is worth noting that the total number of configurations increases quickly with an increased number of configuration values. For example, assigning three configurations to each parameter results in data sets, and adding one extra value to any of the parameters increases the data set count to . The processing speed is dependent on system hardware, but data sets can take hours to process if the number of cycles to failure is high. It is recommended to split up the configurations into multiple smaller sets that can then be run simultaneously through the above-mentioned *ModelShell.m* instances. **Note: It is imperative that they Reynolds number remain below 1\*106. The correlations used fail when the Reynolds value is above this threshold and result in bad data.**

## Running the Model

When the model has been configured as needed, open the *ModelShell.m* file in MATLAB and run the script. The model will first import the configuration data and print to the Command Window the total number of configurations as a confirmation of correct data input, then it will print “Starting iterations…” to indicate that the model is now generating data.

While the model is running, it will output a message whenever it reaches a multiple of 1000 cycles as confirmation that the model is still running and as an indicator of progress on the current configuration.

## Data Output

While the model is running, it will output data from the model at intervals set by the second value in the Settings file. When the model is started, it creates a new folder in the *Outputs* folder with the timestamp as the folder name. A directory is made within this timestamp folder for each configuration named with the indices of the configurations values used from the *ModelConfigurations.xls*. For example, a configuration using the 3rd value for the first parameter and the 1st value for each of the other parameters would have a data output folder named *[3 1 1 1 1 1 1].*

Within the indexed directories, the four types of data output are created as directories for organization: *Damage, Strain, Stress,* and *Temperature*. Data for these types at each node is then saved at the given interval into sequential *.csv* files to be processed into single files later during Preprocessing. Additionally, the *Settings.xls, MaterialParameters.xls,* and *ModelConfigurations.xls* from the *Config* directory are copied in for reference and as a second means of identifying the data sets that were generated.

Once the model has identified that fatigue failure has been achieved, it reports the number of cycles to the Command Window and outputs the last dataset before moving on to the next configuration.

# Preprocessing of Data (Python)

As part of the process of taking the data from the numerical model to the neural network, the data must be processed into a usable format. The output from the numerical model is in *.csv* format, which is usable, but the files are split into distinct files for memory management purposes and also contain far more data than is needed for the neural network, as the assumption is that the material in this setup will fail from the inside out and that the data that can be provided to the neural network in a real application would come from the surface of the material, meaning the state of the most external node is the relevant information that the neural network needs.

The Python-based programs may require some modification to operate correctly. Python is easily modified and the programs are written to place the configuration settings near the top of the relevant file. Editors such as Atom or Spyder are sufficient to view and edit the *.py* files as necessary. Additionally, the preprocessor makes use of the Pandas library, which is included in the recommended Anaconda installation package.

## Checking File Paths are Correct

The preprocessor files *ProcessNumericalModelData\_V3.py* and *PocessInputOutputFiles.py* that are in the *\_Preprocessing* directory in the *Neural Network* directory use absolute file paths to locate the files being processed. Near the top of each file are the directory paths that must be correct for the preprocessor to work.



Figure : Preprocessor directory path variable in ProcessNumericalModelData\_V3.py

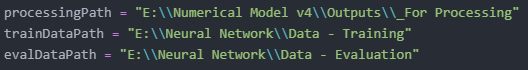


Figure : Preprocessor directory path variables in ProcessInputOutputFiles.py

## Selecting Files for Preprocessing

The first step of this process is to copy the files that are to be preprocessed into the *\_For Preprocessing* directory in the *Outputs* folder of the *Numerical Model v4* folder. The preprocessor files iterate through the directories in this folder and preprocesses them in order, so multiple datasets can be placed in here per preprocessor run.

## Concatenating the Data Files into Summary Files

With the files selected for preprocessing, run the *ProcessNumericalModelData\_V3.py* file. This concatenates all the data files of each type for each set of data provided and then saves the data as a *.csv* file in the Configuration directory for that dataset. When running correctly, the program should print out that it is creating summary files as it progresses through the data.

## Preprocessing for Neural Network Ingest

Once the summary files have been generated, run the *ProcessInputOutputFiles.py* file. The program will prompt the user to ask if the data is for training or evaluation. Enter the number next to the option for the data set and it will assign the outputs to the corresponding folder in the *Neural Network* directory.

Data will be processed into two files in the respective Configuration directory: *Config Inputs.csv,* which contains the input values at each timestep for the neural network, and *Config Output.csv*, which contains the remaining life as a percentage generated from the data. For Training data, the output file is used to tune the network, while the Evaluation data is used to test how well the neural network predicts failure.

Once the second preprocessor file has been executed, data in the *Data* folders can be used for neural network training and evaluation.

# Neural Network (Python)

The Neural Network portion of the ML-TFLAT is composed of a network shell located in the *NN\_Shell.py* file that builds, trains, and tests the networks provided through configuration options set prior to operation. The shell saves network architectures that can later be loaded for re-evaluation or further testing.

## Prerequisites

As mentioned earlier, the network shell depends on several external libraries, but all are included with Anaconda distributions. Keras is the primary library used for the neural network operations, which uses TensorFlow on the backend to perform the necessary tasks for training.

Additionally, Keras can operate in a CPU environment or a GPU environment. For GPU environments, NVIDIA GPUs are required, and this program was developed using an EVGA GeForce 1070 GPU. Note that while the program can run without operating in a GPU environment, taking the time and effort to set one up dramatically reduces the training time required.

## Configuring the Neural Network

In the *Config* folder of the *Neural Network* directory, there are three *.xls* files for configuring the network shell: *Architectures.csv, Build Options.csv,* and *Options.csv.* Additionally, due to how the Python program operates, there is also included a *.txt* file named *ConfigReadme.txt* that contains the information on what options are available at this time for running the network shell.

### Setting Options

The *Options.csv* file contains four rows of information that defines operations of the network shell. The first row defines options for the compiler, the second row provides the parameters for training, the third row parameters are used in evaluation of networks, and the final row describes how data is imported. These values have defined defaults listed in the readme file, but generally they modify how the shell operates.

### Setting Build Options

The *Build Options.csv* file contains options that change how the shell builds networks. These options are described in more detail in the Keras documentation. These options can affect how the networks function and can be used to modify the networks for better precision.

### Setting Network Architectures

The *Architectures.csv* file is used to define network configurations to be built and tested. A network configuration is defined by a series of numbers from left to right in the *Architecture.csv* file, with each new line being a new configuration. The readme file defines how the different numbers identify the different types of layers that can be used to assemble a network configuration. For example, a configuration of 0,1,1,2 would be a network composed of four layers: the 0 represents a Dense layer configured as the input layer to the network, the 1’s represent deep Dense layers that are fully-connected, and the 2 represents a Dense output layer that is configured to be the final layer in the network.

As multiple configurations can be loaded into the *Architectures.csv* file, the network shell can be run once to batch create and train multiple network configurations on the same set of test or evaluation data.

### Setting the Root Path

Like with the preprocessing files, the network shell operates using absolute file paths. For the shell to operate correctly, the root path must be set correctly to the location of the *Neural Network* directory. The rootPath variable is located on line 31 of the *NN\_Shell.py* and can easily be changed based on the location of the directory.



Figure : Root path variable in NN\_Shell.py

### Setting Training/Evaluation Mode

The shell can operate in two different modes: training and evaluation. Training mode builds, trains, then evaluates configurations as per instructions in the *Architecture.csv* file. Evaluation mode loads existing models into the shell and evaluates them based on the contents of the *Data – Evaluation* directory. To change which mode the shell is operating in, the *NN\_Shell.py* file has a Boolean variable and a path array starting on line 25 of the program.

To run in Training mode, set the evalMode variable to False. Training mode does not use the paths in the loadPaths array, so it can be ignored.

To run in Evaluation mode, set the evalMode variable to True and add the absolute paths to the *Model\_Weights.hdf5* files that the shell is to evaluate. What these files are and where to locate them will be discussed in the Shell Outputs section below.

## Using the Network Shell File

Once the shell has been configured, run the *NN\_Shell.py* file. If the model is running in Evaluation mode, a message will be printed to the console saying “Evaluation Only Mode Active”, and a second message, “Evaluation Only Mode Completed!”, will be printed when complete. Both modes provide the evaluation data outputs based on the model’s performance on the currently loaded evaluation data.

In Training mode, the shell will print a message stating that the shell is training models with a percentage completed until all configurations have been built and trained, then will output that it is evaluating models with the percentage completed until all configurations have been evaluated. Once the model has completed all operations, it will print “Evaluation Complete!”

## Shell Outputs

When training a new network, a folder with a sequentially incremented ID number will be created in the *Models* directory for the specific configuration. **Note: The folders do not contain identifying information about the model, so it is important to take note of the ID number in relation to the configuration that built it.** ID numbers are saved in the *Serial Numbers.csv* file in the \_*Documentation* directory. Each model folder is then populated with two directories: *Evaluation* and *Training.* Finally, when the model is saved by the shell, the weights are stored in a file in the model directory called *Model Weights.hdf5.* This is a file format used by Keras to save the weights of each node in a network and can only be loaded into a network of the same architecture as the one that produced the weights.

### Training Outputs

In the Training directory, four files are produced after the training process. *Training Accuracy.csv* contains the accuracy information as reported from Keras in tabular form, with each row containing the accuracy value for each epoch and the columns containing the data for each iteration within the epoch.

*Training Epoch Indices.csv* lists the epoch upon which the shell restarts training as it is designed to drop out and restart training once the model has stopped improving for a set period. This data helps show how the training advances as the model is repeatedly trained. *Training Loss.csv* is like the *Training Accuracy.csv* file except that it stores the loss values instead of the accuracy values.

*Training Info.txt* contains information about the environment variables around the training of the network, including the timestamp at which it is trained, the random seed value that allows repetition of the training in cases where the network uses randomized elements such as node weights, the training time that shows in seconds how long it took to train the model, and the number of failures in training where the network does not improve.

### Evaluation Outputs

The Evaluation directory is populated with five files upon completion of evaluation operations. The *.csv* files are the same as their similarly named counterparts from the Training Outputs, and the two text files provide calculated statistics for the accuracy and loss information that are used to evaluate the performance of each model against others. It is important to evaluate the performance of a model against several parameters and alternative configurations to determine which model has the local maximum effectiveness.

# File Cleanup

When the ML-TFLAT has completed all designated tasks, it is recommended that the data generated from the numerical model be archived to a separate drive or deleted to avoid wasting extensive space on storage medium. The data generated can reach into the gigabyte range on longer simulations and running out of disk space in the middle of generation halts the program. Summary files can be saved to reduce storage bloat as well after preprocessing.

It is also good practice to clear the contents of the *\_For Preprocessing* directory to avoid duplicating preprocessing work. Any models that fail training or underperform should be deleted as well to keep the model space clear, and a directory named ­*\_Valid Models* is provided to help with sorting out effective models.