A Python code for a neural network machine-learning approach for the interpretation of TDS experiments

Nicoletta Marrani[a], Tim Hageman[a,*], Emilio Martínez-Pañeda[a,*]

[a]Department of Engineering Science, University of Oxford, Oxford OX1 3PJ, UK

Abstract

Documentation that accompanies the Python code for the implementation of the neural network machine-learning approach for the interpretation of TDS experiments, available from here. This documentation explains the usage of the machine-learning framework and highlights the main files. Special attention is paid to the parts of the code that implement the model building (e.g., hyperparameter selection) and the model training algorithm. The visualisation of results (i.e., the reconstruction of the TDS spectrum with the predicted trapping parameters and the comparison with experimental data) is also discussed through an example test case.

Contents

---

*Corresponding authors
Email addresses: `tim.hageman@eng.ox.ac.uk` (Tim Hageman), `emilio.martinez-paneda@eng.ox.ac.uk` (Emilio Martínez-Pañeda)

1. Introduction

Understanding hydrogen-material interactions, such as diffusion and trapping, provides insight into the susceptibility of a material to hydrogen embrittlement. Thermal desorption spectroscopy (TDS) is a widely used bulk experimental technique for quantifying the key trapping characteristics of metallic alloys, specifically the trapping capacity, including trap binding energy and density, of different microstructural features (e.g., dislocations, grain boundaries and precipitates). Interpreting the output of TDS experiments, i.e., extracting the number of trap sites, and the binding energy and density associated with each trap, from the TDS spectrum, is far from straightforward. To overcome the limitations of current TDS analysis techniques, we developed a machine learning-based approach, comprising a multi-neural network (NN) machine learning (ML) model trained exclusively on synthetic data, to predict the trapping parameters directly from experimental data. Here, we present a Python implementation of the proposed ML approach. In the remainder of this documentation, we outline the synthetic training data generation, NN model building and training, and required inputs for each test case (e.g., material parameters, TDS test setup, and model hyperparameters). The experimental data preprocessing and model output visualisation will also be discussed. An example test case is provided in Section 4.

1.1. Required external software libraries

In addition to standard Python libraries, the following external software libraries are also required to conduct the analysis:

- Tensorflow (recommended version 2.19.0)

- Keras (recommended version 3.10.0)

Note: Python 3.11.0 is recommended.

1.2. Basic usage

Running "Main.py" performs all actions required for conducting the analysis: It defines the test case-specific inputs and model hyperparameters, checks if the training data and ML models for the given inputs exist and loads them, if they don't exist, it generates the training data and trains the model, performs model validation (i.e., generates validation dataset of 500 data points and plots confusion matrix and predicted vs actual scatter plots for trap density and binding energy), fits the experimental data and plots comparison plots between the model predictions and the experimental data (including the individual trap contributions).

A parameter sweep to compare multiple trained models, differing in the number of training data points, can be carried out by running "TrainMultipleDataSets.py".

2. Summary of included files

The code is set up in an object-oriented manner, defining Python classes for each sub-component and providing accompanying methods. As such, a clear distinction is made between different components, and each can be used and altered with limited/no impact on other components. Here, the different files (and corresponding classes) are described.

2.1. Main.py

This is the main file, from which all classes are constructed, and the actual analysis is performed. Within it, all properties used within the required classes are defined as inputs.

2.1.1. Parameters and setup

- **NumTraining (int)**: Number of samples used for training.

- **NumVerification (int)**: Number of samples used for model verification.

- **Regenerate_Data (bool)**: Flag to regenerate synthetic data.

- **Regenerate_Training (bool)**: Flag to retrain models from scratch.

- **n_cpu_cores (int)**: Number of CPU cores allocated for parallel processing.

- **Traps (str)**: Trap configuration type ("Random" in this case).

- **Concentrations (str)**: Concentration assignment method ("Random").

- **MaxTraps (int)**: Maximum number of traps to predict.

- **ExpName (str)**: Experimental dataset identifier.

- **trap_model (str)**: Trap model selection ("McNabb" or "Oriani").

- **Material**: Material object initialised with experiment and trap model.

- **HyperParameters**: Model hyperparameters loaded from predefined optimised set.

Once all parameters have been defined, the `ModelEnsemble` class, which automates the training data generation as well as the model setup, training, and inference in a unified interface, is initialised with the required inputs:

```
28   # Model creation and training
29   Model = ModelEnsemble.ModelEnsemble(Material, Traps, MaxTraps, Concentrations, HyperParameters,
     ↪   NumTraining, Regenerate_Data, Regenerate_Training, n_cpu_cores)
```

Subsequently, model validation is carried out. Synthetic verification data is generated using `TDS_Sim.SimDataSet` (discussed in Section 2.2.1) and the trained model used to predict traps, concentrations, and energies from the verification data. Plot comparisons between predicted and actual values for traps, concentrations, and energies are provided.

```
31   # Verification
32   TDS_Curves, Actual_Traps, Actual_Concentrations, Actual_Energies, TDS_Temp =
     ↪   TDS_Sim.SimDataSet(Material, NumVerification, MaxTraps, Traps, Concentrations, n_cpu_cores)
33   Predicted_Traps, Predicted_Concentrations, Predicted_Energies = Model.predict(TDS_Curves)
34
35   Model.PlotComparisonTraps(Predicted_Traps, Actual_Traps, TDS_Curves, TDS_Temp)
36   Model.PlotComparisonConcentrations(Predicted_Concentrations, Actual_Concentrations)
37   Model.PlotComparisonEnergies(Predicted_Energies, Actual_Energies)
```

Finally, the experimental TDS data of the corresponding test case ("ExpName") is loaded and processed using the `ExpDataProcessing` class and passed to the trained model as input for the parameter inference.

```
41   # Experimental data fit
42   FileName = 'TDSData/Novak_200_Experimental.xlsx'
43   Exp_Processed_Data = ExpDataProcessing.ExpDataProcessing(FileName, Material, HyperParameters)
44   Exp_Temp = Exp_Processed_Data.Temperature
45   Exp_Flux = Exp_Processed_Data.Flux
46   Exp_TDS_Curve = Exp_Processed_Data.TDS_Curve
47
48   Exp_Predicted_Traps, Exp_Predicted_Concentrations, Exp_Predicted_Energies =
     ↪   Model.predict(Exp_TDS_Curve)
```

The visualisation of the experimental data fit and predicted trap parameter reconstructed curve comparison is performed by the following line:

```
59    # Plot predictions
60    Model.PlotComparisonExpData(Exp_Temp, Exp_Flux, Exp_Predicted_Concentrations,
      ↪   Exp_Predicted_Energies)
```

2.1.3. Additional considerations

- The script uses parallel CPU cores to speed up data generation and training.

- Model training and data generation can be toggled with `Regenerate_Training` and `Regenerate_Data`.

- Energy labelling in the output adjusts according to the selected trap model.

2.2. TDS simulation files

The TDS simulation is carried out by two classes: `TDS_Sim.py` (see Section 2.2.1) and `TDS_Material` (see Section 2.2.2). The `TDS_Sim.py` class is responsible for conducting the FEM-based TDS simulation and for generating the full training dataset. The `TDS_Material` class defines the material properties and test parameters used in `TDS_Sim.py`. It serves as the data storage for material constants, experimental conditions, and numerical setup.

2.2.1. TDS_Sim.py

Initialization:. An instance of this class requires a fully defined `TDS_Material` object, which contains all material properties, numerical settings, and TDS test parameters. The material object is typically initialised from user-defined inputs (via `ExpDataParameters.py`), allowing flexible switching between materials or test setups. Once the material is defined, the simulation functions in `TDS_Sim.py` can be called to perform individual or batch TDS experiments.

The simulation workflow is primarily handled by two functions:

Function: `GenerateDataPoint()`. Generates a single data point by simulating the TDS process under a given set of predefined parameters. It either sets the specified trap energies and concentrations or randomly assigns trapping energies and concentrations, initialises a sample, and performs the three major stages of TDS:

1. **Charging**: Fills the traps with hydrogen (`Sample.Charge()`).
2. **Resting**: Keeps the sample at room temperature for a specified period (`Sample.Rest()`).
3. **TDS Measurement**: Heats the sample and records the desorption signal (`Sample.TDS()`).

Function definition:

```
605   def GenerateDataPoint(i              :int,
606                         Material       :TDS_Material.TDS_Material,
607                         MaxTraps       :int,
608                         NumTraps       :Union[int, str],
609                         Concentration  :Union[float, str]):
```

Core TDS simulation steps:

```
667       Sample = TDS_Sample(Material, N_traps, E_traps, PlotWhileSolving) #initializes material
668       Sample.Charge()     #performs charging
669       Sample.Rest()       #leave at atmospheric pressure
670       [T,J] = Sample.TDS()      #perform TDS
671
```

Function: `SimDataSet()`. Generates a complete dataset by repeatedly invoking `GenerateDataPoint()` in parallel, utilising multiple CPU cores.

Function definition:

```
674   def SimDataSet(Material        :TDS_Material.TDS_Material,
675                  NumVerification :int,
676                  MaxTraps        :int,
677                  NumTraps        :Union[int, str],
678                  Concentrations  :Union[float, str],
679                  n_cpu_cores     :int
680                  ):
```

This function leverages `joblib.Parallel` to distribute the data point simulations across available CPU cores, enabling efficient generation of large datasets.

```
699       Res = joblib.Parallel(n_jobs=n_cpu_cores)(joblib.delayed(GenerateDataPoint)(i, Material,
      ↪   MaxTraps, NumTraps, Concentrations) for i in range(0, NumVerification))
```

2.2.2. TDS_Material.py

This class centralises and organises all input parameters for a TDS simulation, including physical material properties, experiment setup (e.g., heating rates, charging conditions), and numerical discretisation choices.

It extracts these parameters from an external file (e.g., `ExpDataParameters.py`), enabling consistent setup for multiple simulation runs or materials.

The class takes the following as inputs:

- **ExpName** (`str`): A user-defined label or identifier for the experimental dataset. This is used to load the correct parameters via `ExpDataParameters`.

- **trap_model** (`str`): Specifies the trapping model to use. Options include:

  - `"McNabb"` — McNabb-Foster model
  - `"Oriani"` — Oriani's equilibrium trapping model

Upon initialisation, the class pulls the following from `ExpDataParameters`:

- **Material properties:** Lattice site concentration (N_{L}), diffusion energy (`E_Diff`), base diffusivity (`D0`), initial hydrogen concentration (`C0`), trapping rate constant (`TrapRate`), mass density, and molar mass.

- **Experimental setup:** Time and temperature parameters for charging and TDS, sample thickness, heating rate, and atmospheric conditions.

- **Numerical setup:** Time stepping, trap energy resolution, and sample frequency.

2.3. ML model files

The implementation of the ML models is carried out by three classes: `ModelEnsemble` (see Section 2.3.1), `ClassificationModel` (see Section 2.3.2) and `RegressionModel` (see Section 2.3.3).

Both models leverage TensorFlow's Keras API for model training and inference. This is discussed in Section 3.

2.3.1. ModelEnsemble.py

The `ModelEnsemble` class acts as a wrapper for both the classification and regression models. Its primary function is to predict the number of active hydrogen traps in a material and, for the identified number of traps, estimate the corresponding trap energies and concentrations. It automates model setup, training, and inference in a unified interface.

The ensemble structure enables a two-step prediction process:

1. A classification model determines the number of distinct trap types active in a given TDS curve.

2. A corresponding regression model (one per possible number of traps) predicts the associated trap energies and concentrations.

Initialization:. An instance of `ModelEnsemble` requires material properties, training settings, and control flags for regenerating data and retraining models. Depending on whether a fixed number of traps is specified, it either:

- Loads a single regression model (for fixed-trap problems), or

- Loads one regression model per possible trap count and a classification model (for variable-trap problems).

Function: `predict()`. This function accepts a list of processed TDS curves and performs full model inference. It returns:

- The predicted number of traps for each curve.

- Predicted trap energies and concentrations.

```
93          if (isinstance(self.Traps, str)):
94              n_traps = self.ClassModel.predict_traps(TDS_Curve)
95              e, c = self.RegModels[n_traps-1].predict_energies_concentrations(TDS_Curve)
96          else:
97              n_traps = self.Traps
98              e, c = self.RegModel.predict_energies_concentrations(TDS_Curve)
```

Plotting functions:. The class includes several helper functions for visual comparison between predicted and actual outputs:

- `PlotComparisonExpData()`: Overlays model predictions on experimental TDS curves.

- `PlotComparisonTraps()`: Visualizes classification performance and confusion matrix.

- `PlotComparisonEnergies()`, `PlotComparisonConcentrations()`: Compare predicted vs. true regression outputs.

These plots are saved automatically.

2.3.2. ClassificationModel.py

The `ClassificationModel` class defines a neural network classifier that predicts the number of traps present in a material based on its TDS response.

Key attributes:

- `MaxTraps (int)`: Maximum number of traps the model can classify.

- `Concentrations (float or str)`: Fixed or randomised trap site concentration (mol/m^3).

- `NumTraining (int)`: Number of training samples per trap class.

- `SettingsName`, `TrainedModelName (str)`: Filenames for saving/loading model configuration and weights.

- `TrainedModel`: Trained TensorFlow/Keras classification model.

- `n_cpu_cores (int)`: Number of CPU cores allocated for training data generation.

- `Material (TDS_Material)`: Material-specific settings loaded from `TDS_Material.py`.

- `hp (Model_Parameters)`: Hyperparameter object from `Model_Parameters.py`.

Initialisation: The constructor requires simulation parameters and optionally pre-trained regression models. If no saved model is found, or if retraining is explicitly requested, the network is trained from scratch.

```
48        def __init__(self, Material, MaxTraps, Concentrations, HyperParameters, NumTraining,
          ↪  RegModels, Regenerate_Training, n_cpu_cores):
```

```
77            if ((os.path.isfile(self.TrainedModelName) == False) or (Regenerate_Training)):
78                self.Train(RegModels)
79
80            self.TrainedModel = saving.load_model(self.TrainedModelName)
```

Main methods:

`GetData(RegModels)` Retrieves TDS curves and trap count labels from synthetic regression model data. Returns training features and one-hot encoded labels.

`Train(RegModels)` Handles data preprocessing (thresholding, optional log transformation, noise injection), builds a neural network classifier, trains it, and saves the final model.

`predict_traps(TDS_Curve)` Processes a new TDS curve and returns the predicted number of traps.

`PlotConfMatrix(y_test, y_pred)` Generates a normalised confusion matrix to evaluate classification accuracy across trap classes.

Example usage:

```
n_traps = model.predict_traps(new_TDS_curve)
```

The `RegressionModel` class implements a neural network for predicting binding energies and site concentrations for a known number of traps. The outputs depend on the trapping model: Oriani (binding energies) or McNabb-Foster (de-trapping energies).

Key attributes:

- `MaxTraps (int)`: Maximum number of trap sites supported.

- `Concentrations (float or str)`: Fixed or randomised trap concentrations (mol/m^3).

- `NumTraining (int)`: Number of training samples per trap configuration.

- `SettingsName`, `TrainedModelName (str)`: File paths for storing model settings and weights.

- `TrainedModel`: Trained TensorFlow/Keras regression model.

- `n_cpu_cores (int)`: Number of CPU cores allocated for dataset generation.

- `Material (TDS_Material)`: Material input parameters used in simulation.

- `hp (Model_Parameters)`: Hyperparameter set for model training and preprocessing.

Initialisation: The constructor accepts all necessary parameters for training and simulation. If a trained model is not found or retraining is requested, a new model is trained using synthetic data.

```python
60      def __init__(self, Material, MaxTraps, NumTraps, Concentrations, HyperParameters,
        ↪  NumTraining, Regenerate_Data, Regenerate_Training, n_cpu_cores):
```

```python
102             if ((os.path.isfile(self.DataName) == False) or (Regenerate_Data)):
103                 #Data needs to be generated
104                 self.GenerateData()
105
106             if ((os.path.isfile(self.TrainedModelName) == False) or (Regenerate_Training)):
107                 #Train model
108                 self.Train()
```

Main methods:

`GetData(RegModels)` Loads synthetic TDS curves and corresponding continuous output labels (trap energies and concentrations). Returns scaled inputs and targets.

`Train(RegModels)` Applies preprocessing steps (normalisation, optional noise), constructs and trains a deep neural network regression model, and saves the trained weights and output scalers.

`predict_parameters(TDS_Curve)` Accepts a new TDS curve and returns predicted trap parameters (energy and concentration vectors).

`PlotPredictionResults(y_true, y_pred)` Visualises model performance by plotting predicted versus actual trap parameter values.

Example usage:

```
trap_params = model.predict_parameters(new_TDS_curve)
```

2.3.4. Model_Parameters.py

The `Model_Parameters` class encapsulates all hyperparameters used for training both regression and classification models. It supports two modes of initialisation: `"optimised"` (uses predefined, tuned hyperparameters) and `"user_defined"` (allows user input via a parameter dictionary).

Initialisation:

- `ParameterSet (str)`: Specifies the mode (`"optimised"` or `"user_defined"`).

- `UserDefinedParameters (dict, optional)`: Dictionary of custom hyperparameters (currently a placeholder for future extension).

Core attributes:

- `flux_threshold (float)`: Minimum flux value for numerical stability (default: `1e-12`).

- `log_transformation (bool)`: Whether to apply a base-10 log transformation to input data (default: `True`).

- `noise_std_dev (float)`: Standard deviation of Gaussian noise added during data augmentation (default: `0.05`).

- `epoch_coeff_reg (int)`: Multiplier for the number of training epochs in regression models (default: `200`).

- `epoch_coeff_class (int)`: Multiplier for training epochs in classification models (default: `100`).

- `batch_size (int)`: Number of samples per training batch (default: `32`).

Optimised hyperparameters:

- **Regression model:**

  - Number of layers: `5`

  - Nodes per layer: `[64, 64, 32, 16, 8]`

  - Learning rate: `1e-3`

– Weight decay: `1e-3`

- **Classification model:**

  – Number of layers: `4`

  – Nodes per layer: `[256, 128, 64, 32]`

  – Learning rate: `1e-3`

  – Weight decay: `1e-3`

2.4. Experimental data files

2.4.1. ExpDataParameters.py

The `ExpDataParameters` class stores material properties, experimental conditions, and numerical settings for specific datasets. Upon initialisation, it assigns all relevant attributes based on the input `ExpName`.

Creating an instance of this class using a valid `ExpName` assigns all relevant parameters required for experimental data processing and simulation.

2.4.2. ExpDataProcessing.py

The `ExpDataProcessing` class handles preprocessing of raw experimental TDS data to generate uniformly sampled, smoothed input compatible with machine learning model training.

Initialisation:

The class is initialised with the following inputs:

- `FileName`: Path to the Excel file containing raw experimental data.

- `Material`: A fully defined `TDS_Material` object containing material-specific parameters.

- `HyperParameters`: A `Model_Parameters` object containing preprocessing settings.

Upon instantiation, the following methods are automatically called:

1. `load_and_process_data()`
2. `smooth_and_downsample_data()`

Methods:

load_and_process_data().

Loads the raw experimental data from an Excel file. The first column is the temperature (in K), and the second column contains the desorption rate.

smooth_and_downsample_data().

This method processes the raw data in the following steps:

1. Scale the desorption rate using material thickness and density.

2. Apply Savitzky-Golay filtering to smooth the signal.

3. Interpolate the smoothed data onto a uniform temperature grid (`ntp` steps) using cubic interpolation.

4. Clip desorption values above the maximum temperature to a defined flux threshold.

5. Store the final processed arrays as model input.

Output attributes:

The processed outputs are stored as attributes for downstream use:

- `self.Temperature`: Uniformly spaced temperature values excluding the final temperature point (`TMax`).

- `self.Flux`: Corresponding desorption rates after smoothing and interpolation.

- `self.TDS_Curve`: A TensorFlow tensor containing the processed data, ready to be used as model input.

3. Specifics: NN building and training

In both `ClassificationModel` and `RegressionModel` classes, the model construction and training are handled by the `Train()` function. This function is only called if the model has not been previously trained (i.e., no saved model file is found) or if training is explicitly requested using the `Regenerate_Training` flag. This is done for the regression model as:

```
106     if ((os.path.isfile(self.TrainedModelName) == False) or (Regenerate_Training)):
107         #Train model
108         self.Train()
```

And for the classification model as:

```
77      if ((os.path.isfile(self.TrainedModelName) == False) or (Regenerate_Training)):
78          self.Train(RegModels)
```

The classification model's `Train()` function takes the regression models as input in order to access the training data. Specifically, the regression model defines the filenames for the input data, which are retrieved via the `GetData()` function.

Within both classes, the training process follows the same four stages:

1. Preprocessing the training data

2. Train-test splitting

3. Model building

4. Model training

A detailed breakdown of each stage (for the classification model) is given below.

I. Preprocessing training data. This stage involves three key preprocessing steps:

- **Flux Thresholding:** Small values are clipped for numerical stability.

```
127        All_TDS = np.where(All_TDS < self.hp.flux_threshold, self.hp.flux_threshold,
           ↪  All_TDS)
```

- **Log Transformation:** Optionally applies a base-10 logarithmic transformation.

```
128        if self.hp.log_transformation:
129            All_TDS = np.log10(All_TDS)
```

- **Noise Augmentation:** Adds independent Gaussian noise to each feature for robustness.

```
132        All_TDS_noise = All_TDS.copy()
133        selected_features = range(All_TDS.shape[1])
134        for feature_idx in selected_features:
135            noise = np.random.normal(0, self.hp.noise_std_dev, All_TDS.shape[0])
136            All_TDS_noise[:,feature_idx] += noise
```

II. Train-test splitting. The preprocessed data is split into training and test sets. The test set size is the minimum of 200 samples or 20% of the dataset, scaled by the number of traps:

```
139        TestsetSize = min(200, math.floor(0.2*self.NumTraining))*self.MaxTraps
140        X_train, X_test, y_train, y_test = train_test_split(All_TDS_noise, Labels,
           ↪  test_size=TestsetSize)
```

III. Model building. The model architecture consists of an input normalisation layer, a series of hidden layers, and a softmax output layer. The number of hidden layers and their sizes are determined by hyperparameters.

```
145          # Build model
146          NormLayer = layers.Normalization()
147          NormLayer.adapt(X_train)
148          model = models.Sequential()
149          model.add(layers.Input(shape=(X_train.shape[1],)))
150          model.add(NormLayer)
151
152          # Add Dense layers based on num_layers
153          for i in range(0,self.hp.num_layers_class):
154              model.add(layers.Dense(self.hp.nodes_class[i]*self.MaxTraps,
             ↪   kernel_initializer=initialiser))
155              model.add(layers.ReLU())
156
157          # Add Output layer
158          model.add(layers.Dense(self.MaxTraps, activation="softmax"))
159
160          # Compile model
161          model.compile(optimizer=tf.keras.optimizers.Adamax(learning_rate=self.hp.lr_class,
             ↪   weight_decay=self.hp.wd_class),
162                       loss='categorical_crossentropy')
```

IV. Model training. The model is trained using `model.fit()`. Training hyperparameters are defined in the `Model_Parameters` class and are passed during class initialisation.

```
164          # Train model
165          model.fit(x=X_train, y=y_train,
166                              validation_data=(X_test, y_test),
167                              batch_size=self.hp.batch_size,
168                              epochs=self.hp.epoch_coeff_class*self.MaxTraps,
169                              verbose=1)
```

The regression model follows the same general pipeline, with one key distinction: it applies separate normalisation (i.e., scalers) to the energy and density outputs using two independent `MinMaxScaler` instances.

```
189        SolScalerEnergy = MinMaxScaler(feature_range=(0,1))
```

Both scalers are then saved separately.

4. Sample results

In its current setup, running `Main.py` will conduct the analysis for the experimental data obtained by Novak et al. [1], corresponding to a high-strength AISI tempered martensitic steel ($\phi = 200°\text{C/h}$), i.e., test case 1 of the manuscript.

The test case-specific parameters (i.e., material parameters, TDS setup configuration and numerical parameters) have been included in the `ExpDataParameters` class under "Novak_200". The optimised hyperparameters (as described in the manuscript) have been included in the `Model_Parameters` class and can be selected by setting the `ParameterSet` variable as "optimised".

First, all parameters required for the analysis are specified, and the material, hyperparameters (all other classes) are initialised. Subsequently, the analysis is carried out as described in Section 2.1.

Parameter definition:. The test case is set as "Novak_200", and the McNabb-Foster framework is specified as the trap model. The material can then be defined/initialised (see `Main.py`)

Data generation and model training:. The data generation, model building and training, model evaluation and experimental data pre-processing and fit are carried out as described in Section 2.1.

Results and figures:. Several figures are produced and saved to the *Figures* folder in the folder specific to the test case, i.e., *Novak_200* folder. Six figures are saved: the confusion matrix, scatter plot predicted vs actual for trap density, scatter plot predicted vs actual for binding energy, the wrongfully predicted TDS curves, plot of the experimental data vs the reconstructed TDS spectrum based on the model predictions, and the reconstructed TDS spectrum with the trap contributions (i.e., individual peak corresponding to each trap). The first four figures correspond to the model validation. The latter two figures are the experimental data fit. These are shown in Fig. 1.

Additionally, the values of binding/de-trapping energy and the trap density of each trap are printed in the terminal. Note: the energies are reported as absolute values in ascending order.

If the McNabb-Foster framework is used, the values appear as:

```
Number of traps: 4
Trap 1: De-trapping energy = 73789.01 J/mol, Trap density = 8.95 mol/m^3
...
```

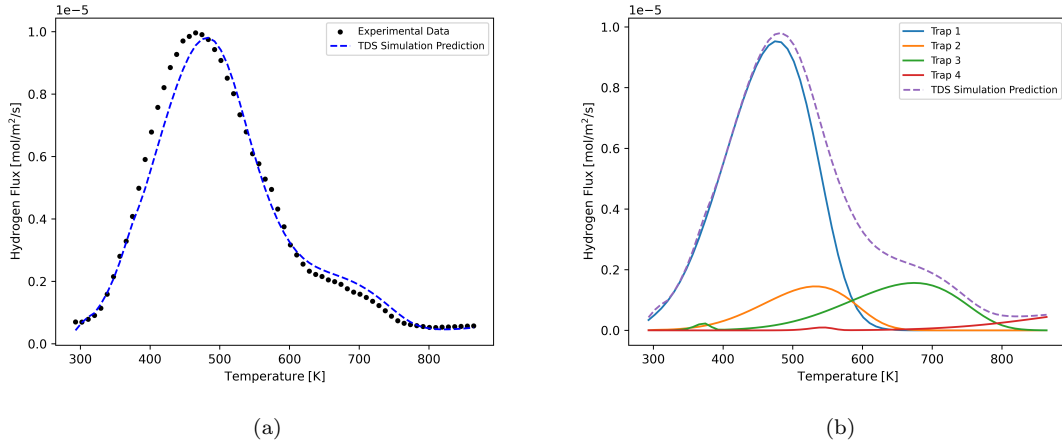Conversely, if the Oriani framework is used, the values appear:

Figure 1: Results produced from running "Main.py", for the high-strength AISI tempered martensitic steel ($\phi = 200°\text{C/h}$), corresponding to test case 1 of the manuscript or "Novak_200" in "ExpDataParameters.py". A comparison between the reconstructed TDS spectrum based on the ML model predictions and the experimental data is provided in (a). The individual trap contributions are depicted in (b).

```
Number of traps: 4
Trap 1: Binding energy = 53789.01 J/mol, Trap density = 8.95 mol/m^3
...
```

References

[1] P. Novak, R. Yuan, B.P. Somerday, P. Sofronis, and R.O. Ritchie. A statistical, physical-based, micromechanical model of hydrogen-induced intergranular fracture in steel. *Journal of the Mechanics and Physics of Solids*, 58(2):206–226, February 2010. Publisher: Elsevier BV.