Code for Domenico et al. (2023)

Saturday, February 17, 2024 11:12 AM

----
dataset.py:
----

extends the `Dataset` class from PyTorch, primarily used for handling and preprocessing fermentation data for machine learning models

Class Definition: `FermentationData(Dataset)`

a. **`__init__`** method: Initializes the dataset object with parameters like working directory, training mode flag, output variables, window size (**`ws`**), and stride for windowing.
    - **`work_dir`**: Directory where the data is stored.
    - **`train_mode`**: Flag to indicate whether the dataset is for training or testing.
    - Lists **`train_fermentations`** and **`test_fermentations`** define the indices of fermentation batches used for training and testing, respectively.
    - **`cumulative_var`** and **`binary_var`** list variables in the dataset that are cumulative and binary, which will be treated differently during preprocessing.
    - **`x_var`** and **`y_var`** define input and output variables for the model.
    - Normalization parameters are calculated using a specific fermentation batch (number 22) for standardizing the data.
    - Data is loaded and preprocessed using internal methods, tailored for training or testing based on the **`train_mode`** flag.

b. **`load_data`** method: Loads the fermentation data based on given indices. It can load data for a single fermentation or multiple based on the training/testing mode. Utilizes the **`utils.load_data`** function, indicating the use of an Excel file (**`data.xlsx`**) as the data source.

c. **`preprocess_data`** method: Applies several preprocessing steps to the input data **`X`**, such as converting cumulative data to snapshot data, normalizing the data (default using z-score normalization), and segmenting data into sequences using a sliding window approach.

d. **`preprocess_labels`** method: Similar to **`preprocess_data`**, but applied to the output labels **`Y`**. It includes an interpolation step (likely to fill missing values or smooth the data) before segmenting the data into sequences.

e. **`cumulative2snapshot`**, **`normalise`**, **`data2sequences`**, **`polynomial_interpolation`**, **`linear_local_interpolation`**, and **`mix_interpolation`** methods: These are helper methods used within preprocessing to transform the data appropriately. For example, **`cumulative2snapshot`** converts cumulative data points into individual measurements, and **`normalise`** standardizes the data.

f. **`__getitem__`** method: Overrides the method from the PyTorch **`Dataset`** class to return a single training/testing example. It converts the selected data point into PyTorch tensors before returning.

g. **`__len__`** method: Also overrides the PyTorch **`Dataset`** class method to return the total number of examples in the dataset.

h. **`get_num_features`** method: Returns the number of features in the input data, useful for defining model architectures that depend on the input size.

This class effectively encapsulates the entire process of data loading, preprocessing, and formatting necessary for training and evaluating machine learning models on fermentation data, particularly designed for recombinant protein optimization studies.

Useful tips:

Based on the provided **`Dataset.py`** code, the **`FermentationData`** class indeed loads data for each specified fermentation batch as a pandas DataFrame and then appends each DataFrame's values to lists **`X`** and **`Y`**. This approach is implemented in the **`load_data`** method within the class.

Here is a breakdown of the process:

Initialization: In the **`__init__`** method of **`FermentationData`**, it initializes lists **`self.train_fermentations`** and **`self.test_fermentations`** with the batch numbers intended for training and testing, respectively.

Data Loading: The **`load_data`** method is called twice during initialization - once to set normalization parameters using a specific fermentation batch (in this case, **`self.fermentation_norm_number = 22`**), and again to actually load the data for the specified training or testing batches.

Batch Data Loading: For each batch number in **`self.train_fermentations`** or **`self.test_fermentations`**, the method loads data using the **`utils.load_data`** function. This function is expected to read the Excel file for the specified batch and return the data as numpy arrays (or possibly pandas DataFrames that are then converted to numpy arrays).

Appending Data: The numpy arrays for each batch are appended to lists **`X`** and **`Y`**. If **`X`** and **`Y`** are indeed lists of DataFrame values, then after appending, they would look something like this: **`X = [array_from_df1, array_from_df2, array_from_df3, ...]`** and similarly for **`Y`**.

Conversion to Numpy Arrays: At the end of the **`load_data`** method, **`X`** and **`Y`** are converted to numpy arrays using **`np.array(X)`** and **`np.array(Y)`**, resulting in a structure like **`np.array([ [df_1_values], [df_2_values], [df_3_values], ... ])`**. This creates a 3D numpy array where each "sub-array" corresponds to the data from one batch.

Preprocessing: After loading, the data is preprocessed (normalized, converted from cumulative to snapshot data, interpolated, and converted into sequences) before being used in training or testing.

Conversion to Tensors and Indexing: In the **`__getitem__`** method, individual items (sequences) from **`X`** and **`Y`** are converted to PyTorch tensors, which makes the dataset compatible with PyTorch's data loading and training mechanisms.

This approach allows for the aggregation of data from multiple fermentation batches into a unified dataset structure, where each element in the dataset corresponds to a sequence of data points from the time series of a batch. The model can then be trained on this aggregated data, learning from the patterns across all included batches.

----

data_analysis.py:

The **`data_analysis.py`** script is designed for processing and visualizing data related to fermentation, focusing on variables that affect and indicate the process's efficiency, such as pH, oxygen levels, temperature, and the optical density at 600 nm (OD600),

Variables:

- **`x_var`**: Lists input variables related to the fermentation process.
- **`y_var`**: Specifies the output variable, OD600, used to measure the fermentation process's efficiency.

Functions:

- **`mix_interpolation`**: Applies a mixed interpolation method to handle missing data points in the dataset, using a combination of linear local interpolation strategies.
- **`data2sequences`**: Converts continuous data into sequences using a sliding window approach, facilitating the analysis of time-series data.
- **`preprocess_labels`**: Applies preprocessing steps to the output labels, including interpolation and conversion into sequences, to prepare the data for analysis or modeling.
- **`get_interpolation`**: Specifically focuses on interpolating the output labels, providing both the interpolated data and its sequential representation.
- **`unique`**: Extracts unique elements from a list, useful for data cleaning or analysis tasks.

Data Analysis and Plotting:

- The script sets up directories for storing analysis results and iterates through specified fermentation batches to load and plot data for each input variable (**`x_var`**).
- For each variable, data from different batches are loaded, potentially downsampled for specific variables (**`m_stirrer`**, **`m_ls_opt_do`**), and plotted to visualize trends and variations across batches.
- Separate sections of the script are dedicated to plotting OD600 values for all specified batches, highlighting the growth characteristics of each batch in a semilogarithmic plot to better visualize exponential growth phases.
- An additional plotting section focuses on demonstrating the interpolation of OD600 values for a specific batch, comparing actual values with interpolated ones to assess the interpolation method's efficacy.

Definitions to keep in mind:

a. Interpolation:

   Interpolation is a method used to estimate unknown values within the range of a discrete set of known data points. In time-series data, there might be missing timestamps or measurements. Interpolation helps fill in these gaps by constructing new data points within the range of a discrete set of known data points, making the dataset more complete and suitable for analysis.

   `mix_interpolation` function seems to use a mixed strategy, possibly combining multiple interpolation methods to handle missing data in the fermentation process more effectively.

b. Conversion into Sequences:

   Conversion into sequences, or "windowing," is a technique used to transform continuous time-series data into a series of overlapping or non-overlapping segments or sequences. This is particularly useful in machine learning and signal processing, where fixed-length inputs are required for models or algorithms

   `data2sequences` function likely implements this technique, preparing the data for time-series analysis or machine learning models that require input data in fixed-length sequences.

----

model.py:

Common Components:

- **Imports**: The script imports necessary modules from PyTorch, specifically **`torch`** and **`nn`** (neural networks). The **`pdb`** module is also imported for debugging purposes.
- **Initialization (`__init__`)**: Both models are initialized with similar parameters:
  - **`input_dim`**: The number of features in the input data.
  - **`hidden_dim`**: The size of the hidden layers within the LSTM/RNN cells.
  - **`output_dim`**: The number of features in the output data, or the prediction size.
  - **`n_layers`**: The number of stacked LSTM/RNN layers.
- **Pre-FC Layer (`pre_fc`)**: A fully connected (linear) layer that transforms the input dimension to the hidden dimension before feeding it into the LSTM/RNN layers. This can help in learning a better representation of the input data.

- **Activation Function (`relu`)**: The ReLU (Rectified Linear Unit) activation function is used to introduce non-linearity into the model, allowing it to learn more complex patterns.

LSTMPredictor:
- Utilizes an LSTM (Long Short-Term Memory) layer, suitable for learning from sequences with long-range dependencies. LSTMs are a type of recurrent neural network (RNN) capable of remembering information for long periods, which is crucial for time-series data.
- **Forward Pass (`forward` method)**: Defines how the data flows through the model. The input data is first passed through the **`pre_fc`** layer and ReLU activation, then through the LSTM layer followed by another ReLU activation, and finally through a fully connected layer (**`fc`**) to get the output.
- **Hidden State Initialization (`init_hidden` method)**: Initializes the hidden and cell states for the LSTM with zeros. This is necessary as LSTMs maintain a hidden state and a cell state across sequences.

RNNpredictor:
- Similar in structure to **`LSTMPredictor`** but uses a simpler RNN (Recurrent Neural Network) layer instead of LSTM. RNNs are also suited for sequence data but might struggle with longer dependencies due to issues like vanishing gradients.
- **Forward Pass (`forward` method)**: Similar to **`LSTMPredictor`**, but the data is passed through an RNN layer instead of an LSTM layer.
- **Hidden State Initialization (`init_hidden` method)**: Initializes the hidden state for the RNN with zeros. Unlike LSTMs, standard RNNs only maintain a single hidden state.

----
train.py:

The train.py script is designed for training and evaluating machine learning models, specifically focusing on time-series data from fermentation processes, using PyTorch.

Setup and Argument Parsing:
- Imports necessary libraries and modules for deep learning (**`torch`**, **`torch.nn`**, **`torch.optim`**), data manipulation (**`numpy`**), and visualization/logging (**`wandb`** for Weights & Biases).
- **`argparse`** is used to parse command-line arguments such as batch size, learning rate, hidden dimensions, number of layers, and epochs, which allows for flexible experimentation.
- Initializes random seeds for reproducibility and sets the GPU device based on the provided argument.

Training Function (`train`):
- Sets the model to training mode using **`model.train()`**.
- Iterates over the training data loader, fetching batches of inputs and labels.
- Initializes the model's hidden state for each batch and performs a forward pass through the model.
- Computes the loss using a custom **`compute_loss`** function (likely Mean Squared Error - MSE) and backpropagates errors to update the model parameters.
- Logs training loss and error metrics periodically and optionally to Weights & Biases (wandb) for tracking experiments.

Testing Function (`test`):
- Evaluates the model in inference mode using **`model.eval()`**.
- Similar to the training loop, it iterates over the test data loader, but gradients are not computed to save memory and computation (**`torch.no_grad()`**).
- Computes and logs the loss and Root Mean Squared Error (RMSE) for the test data, providing an evaluation metric for model performance.

Loss Computation (`compute_loss`):
- Defines a loss function, which is likely the Mean Squared Error (MSE) between the model outputs and the true labels, a common choice for regression tasks.

Data Preparation:
- Instantiates training and testing datasets using the **`FermentationData`** class from **`dataset.py`**, which likely preprocesses the fermentation data for model consumption.
- Data loaders are created for both training and testing datasets, facilitating batch-wise data loading during model training and evaluation.

Model Initialization:
- Depending on the command-line argument, initializes either an **`LSTMPredictor`** or an **`RNNpredictor`** model with specified configurations (input dimension, hidden dimension, output dimension, number of layers).
- Prepares the model for training on a GPU if available.

Optimizer:
- Sets up an optimizer, specifically Stochastic Gradient Descent (SGD) with a defined learning rate and weight decay, for updating model parameters during training.

Training Loop:
- Executes the training process for a specified number of epochs, calling the **`train`** and **`test`** functions in each epoch to train the model and evaluate its performance on the test set.

- Tracks and saves the model with the best performance based on RMSE, allowing model checkpointing for later use or further evaluation.

Weights & Biases Integration:
- Optionally integrates with Weights & Biases for experiment tracking and visualization, logging training and validation metrics for each epoch, and keeping track of the best RMSE achieved.

This script encapsulates the entire workflow for training, evaluating, and tracking the performance of LSTM or RNN models on time-series data, making it a comprehensive tool for deep learning experiments in fermentation process optimization or similar domains.

----
test.py:

The **`test.py`** script is designed for testing machine learning models on a dataset, particularly focusing on time-series data from fermentation processes. It involves loading a trained model, running it on a test dataset, and evaluating its performance. Here's a detailed explanation of the script's components:

Setup and Argument Parsing:
- The script begins by importing necessary libraries and setting up the environment. **`argparse`** is used to parse command-line arguments, allowing users to specify parameters such as batch size, hidden dimensions, number of layers, and the path to the trained model weights.
- Random seeds are set for reproducibility, and the GPU device is specified.

Test Function (`test`):
- The **`test`** function is designed to evaluate the model on the test dataset. The model is set to evaluation mode using **`model.eval()`**.
- The function initializes storage for predictions, labels, and a count of overlaps (since sequences may overlap in time-series data) to correctly average predictions over overlapping regions.
- Iterates through the test data loader, processing each batch. For each input batch, it initializes the hidden state, performs a forward pass through the model to get predictions, and optionally applies smoothing to the predictions using a moving average.
- Accumulates the sum of predictions and labels in overlapping regions and counts the overlaps to later compute the average prediction per time step.
- Calculates loss and error (RMSE - Root Mean Squared Error) for the entire dataset.
- Returns the error, along with the predictions and labels for further analysis.

Data and Model Preparation:
- Loads the test dataset using the **`FermentationData`** class, specifying that it's for testing (not training) and the output variable of interest (**`od_600`**).
- Initializes the model (either **`LSTMPredictor`** or **`RNNpredictor`** based on the command-line argument) with the appropriate dimensions and layers.
- Loads the trained model weights from the specified path.

Model Evaluation:
- Calls the **`test`** function to evaluate the model on the test dataset, capturing the error, predictions, and labels.
- Post-processes the predictions and labels by trimming the initial entries (first 50 in this script) to remove potential edge effects from the initial hidden state or sequence padding.

Performance Metrics and Results Saving:
- Calculates the RMSE between the predictions and labels as a measure of model performance.
- Computes the Relative Error on Final Yield (REFY) as the percentage error of the model's final prediction compared to the final true label, providing insight into the model's accuracy at predicting the final outcome of the fermentation process.
- Saves the predictions, labels, RMSE, and REFY to an **`.npz`** file for later analysis or visualization.
- Calls a utility function **`plot_od600_curve`** to generate a plot comparing the predicted and true OD600 curves, annotated with the RMSE and REFY metrics.

----
utils.py:

The **`utils.py`** script provides a collection of utility functions for data loading, preprocessing, normalization, interpolation, sequence transformation, plotting, and model weight management. These functions are designed to support the processing and analysis of time-series data, such as fermentation data, and to facilitate the training and testing of machine learning models. Here's a breakdown of the key functions and their purposes:

Data Loading and Preprocessing:
- **`load_data`**: Loads data from Excel files based on specified input (**`x_cols`**) and output (**`y_cols`**) variables. It converts the data into numpy arrays for further processing.
- **`cumulative2snapshot`**: Converts cumulative data into individual snapshot values, useful for variables that accumulate over time, such as total volume of added substances.
- **`get_norm_param`**: Calculates the mean and standard deviation of the dataset, which are used for normalization.
- **`z_score`**: Applies z-score normalization to the data using the calculated mean and standard deviation, standardizing the data to have a mean of 0 and a standard deviation of 1.

- **normalise**: General function for data normalization, calling **z_score** when the mode is set to "z-score".

Sequence Transformation:
- **data2sequences**: Transforms continuous data into sequences using a sliding window approach, which is essential for preparing time-series data for recurrent neural network models.
- **compute_padded_length**: Calculates the necessary padding to make the data fit into the specified window size and stride, ensuring all data is included in the sequence transformation.

Interpolation:
- **polynomial_interpolation** and **linear_local_interpolation**: Provide two methods for interpolating missing data points in a time series, using polynomial and linear approaches, respectively.
- **mix_interpolation**: Combines linear and polynomial interpolation methods to fill in missing data points, potentially offering a more robust interpolation by leveraging the strengths of both methods.

Plotting and Visualization:
- **plot_od600_curve**: Generates a plot comparing predicted OD600 values against actual (interpolated) values, annotating the plot with performance metrics like RMSE (Root Mean Squared Error) and REFY (Relative Error on Final Yield).

Model Weight Management:
- **save_weights**: Saves the trained model weights and the epoch number to a file, allowing for checkpointing and model persistence.
- **load_checkpoint** and **load_weights**: Functions for loading model weights from a checkpoint file, facilitating model evaluation or further training.

Miscellaneous:
- **reject_outliers**: Excludes outliers from the dataset based on a specified threshold, helping to clean the data and improve model performance.
- **sliding_window_view**: Utilized within **data2sequences** for generating sliding windows over the data, likely imported from **numpy.lib.stride_tricks**.

This script is a crucial component of the larger machine learning pipeline, providing the necessary tools for data manipulation, preprocessing, and evaluation, thereby supporting the end-to-end process of training, testing, and analyzing models for time-series forecasting tasks.

----

plot_results.py:

The **plot_results.py** script is designed for visualizing the results of machine learning models, specifically comparing the performance of LSTM and RNN models on a given dataset, likely related to fermentation processes. The script generates plots that show the predicted optical density at 600 nm (OD $_{600nm}$) by both models against the ground truth for a specified batch. Here's an overview of how the script works:

Argument Parsing:
- The script uses **argparse** to parse command-line arguments. It expects an optional argument **-b** or **--batch** to specify the batch number for which the results will be plotted.

Plotting Function (save_plot):
- The function **save_plot** takes a file path and data from LSTM and RNN models as input.
- It initializes a figure and plots the predictions made by the LSTM and RNN models using a semilogarithmic plot (**plt.semilogy**), which is useful for visualizing data that spans several orders of magnitude, as is often the case with OD measurements in fermentation.
- The ground truth data is also plotted with a dashed line for comparison.
- The plot is customized with labels for the x-axis (**timestamp**) and y-axis (**OD$_{600nm}$**), and the legend is positioned in the lower right corner of the plot.
- The **ticklabel_format** function is used to format the x-axis labels in scientific notation, which is useful for time-series data that can have large numerical values.
- The plot is saved as a PNG file at the specified file path, with a resolution of 600 DPI for high-quality output.
- After saving the plot, the script prints a confirmation message along with the Relative Final Yield Error (REFY) for both LSTM and RNN predictions. REFY is calculated as the percentage difference between the final prediction and the final ground truth value, providing a measure of the model's accuracy at predicting the final outcome of the fermentation process.

Data Loading and Plot Generation:
- The script constructs the file paths for the results of the LSTM and RNN models based on the provided batch number and a predefined directory path (**dir_path**).
- It loads the results data from **.npz** files for both LSTM and RNN models using **numpy.load**.
- The **save_plot** function is then called with the constructed file path for saving the plot and the loaded data for both models to generate and save the comparison plot.

This script is a useful tool for visually comparing the performance of different models on time-series data, such as OD measurements from fermentation processes, allowing for a clear visual representation of model accuracy and prediction capabilities.

Another useful tips:

Based on the code files you've provided, here's how the data was loaded and used in the deep learning model:

Data Loading Process:

- **Data Source**: The data for each fermentation batch is stored in separate Excel files within folders named after the batch numbers (e.g., 8, 11, 12, 14, 16, etc.).

- **Data Extraction**: The `load_data` function in the `utils.py` script is responsible for loading data from these Excel files. This function takes parameters such as `work_dir` (directory path), `fermentation_number` (batch number), `data_file` (Excel file name), and `x_cols`, `y_cols` (columns to be used as input and output variables, respectively).

- **Data Transformation**: The data extracted from Excel files is then transformed into numpy arrays, with separate arrays for input variables (`X`) and output variables (`Y`).

Data Usage in Model:

- **Single Model for All Batches**: The code indicates that a single deep learning model (either LSTM or RNN, based on the user's choice) is used for all the batches, rather than separate models for each batch.

- **Dataset Preparation**: The `FermentationData` class, defined in `Dataset.py`, is utilized to create dataset instances for both training and testing. This class handles data loading for specified batches (using the `load_data` function), data preprocessing (such as normalization and sequence transformation), and preparing the data in a format suitable for the deep learning model.

- **Batch-wise Data Handling**: During the instantiation of `FermentationData` objects, lists of fermentation batch numbers are specified for training and testing. This indicates that the model does not train on each batch separately but rather on a collection of batches specified for training. Similarly, the model is tested on a separate set of batches.

- **Combining Data for Training and Testing**: The data from multiple batches specified for training is loaded, preprocessed, and combined into a single dataset. This combined dataset is then used to train the model. A similar process is followed for the test dataset.

- **Tensor Dataset and DataLoader**: The preprocessed and combined data is converted into a PyTorch `Dataset` object (`FermentationData`), which is then passed to a PyTorch `DataLoader`. The `DataLoader` handles batching and shuffling of the data, making it ready for input into the deep learning model.

- **Model Training and Testing**: The `train.py` and `test.py` scripts handle the training and testing of the model, respectively. In the training script, the model is trained on the combined training dataset, and in the testing script, the model's performance is evaluated on the combined test dataset.