Signal Classification

1. Introduction

This project focuses on classifying vibration signals from industrial machines into healthy or faulty categories using deep learning. It includes two main approaches:

- 1D CNN applied to raw time-series signals
- 2D CNN applied to spectrograms derived from the signals

The goal is to build an automated pipeline for fault detection using signal processing and neural networks.

2. Prerequisites

- **Programming Language:** Python 3.11+
- **Libraries:**
  - `numpy==2.0.2`
  - `pandas==1.5.3`
  - `scikit-learn==1.6.1`
  - `tensorflow==2.18.0`
  - `scipy==1.15.3`
  - `matplotlib==3.10.0`

Install dependencies using:

pip install -r requirements.txt

3. Project Structure

```
signal_classification/
├── data/                  # Raw and processed signal data
│   ├── raw/
│   │   ├── Normal Condition/   # Healthy signals (.txt)
│   │   └── Looseness/          # Faulty signals (.txt)
│   └── Looseness/          # Faulty signals (.txt)
├── models/                # Trained models (.h5 files)
├── notebooks/             # Jupyter notebooks for training and evaluation
├── src/                   # Source code modules
│   ├── preprocessing.py
│   ├── train.py
│   └── evaluate.py
├── requirements.txt       # Python dependencies
└── README.md              # This file
```

4. Code Documentation

4.1. Preprocessing Script (`src/preprocessing.py`):

"""

This script handles data cleaning, normalization, and feature engineering.

Functions:

- load_signal_and_fs(path): Load a vibration signal and compute its sampling frequency from a .txt file.
- signal_to_spectrogram(signal, fs, nperseg=None, noverlap=None, target_shape=(128, 128)): Convert a 1D time-domain signal to a 2D spectrogram image.
- load_dataset(healthy_dir, faulty_dir, output_pkl_path):Load and preprocess all signals from given directories and convert them to spectrograms.
- load_signal(filepath):Load a time-domain signal from a text file.

- load_all_signals(healthy_dir, faulty_dir, output_pkl_path):Load and label all signal files from healthy and faulty directories.
- fix_signal_lenght(signal, desired_length=25000):Adjust the length of a 1D signal to a fixed size by trimming or padding.
- normalize_signal(signal):Normalize a 1D signal using Min-Max scaling to the [0, 1] range.

"""

```
def load_signal_and_fs(path):
    """
    Load a vibration signal and compute its sampling frequency from a .txt file.

    The function reads header metadata (such as Max_X and NoOfItems) to extract
    the total duration and number of samples, and uses them to compute the sampling rate.
    It also loads the signal values, ignoring comment lines starting with '%'.

    Args:
        path (str): Path to the .txt file containing the signal and metadata.

    Returns:
        tuple:
            - signal (np.ndarray): The loaded 1D signal array.
            - fs (float): The computed sampling frequency (samples per second).

    Raises:
        ValueError: If the necessary metadata (duration or number of samples) cannot be extracted.
    """


def signal_to_spectrogram(signal, fs, nperseg=None, noverlap=None, target_shape=(128, 128)):
    """
    Convert a 1D time-domain signal to a 2D spectrogram image.
```

```
    Applies Short-Time Fourier Transform (STFT) to extract time-frequency features.
    The spectrogram is then log-scaled, normalized to [0, 1], and resized to a fixed shape.

    Args:
        signal (np.ndarray): Input 1D signal array.
        fs (float): Sampling frequency of the signal.
        nperseg (int, optional): Window size for STFT. If None, defaults to fs / 5.
        noverlap (int, optional): Overlap between windows. Defaults to nperseg / 3.
        target_shape (tuple): Desired output size for the spectrogram (height, width).

    Returns:
        np.ndarray: 2D normalized and resized spectrogram image.
    “””


def load_dataset(healthy_dir, faulty_dir, output_pkl_path):
    """
    Load and preprocess all signals from given directories and convert them to spectrograms.

    For each .txt signal file, this function:
    - Loads the signal and computes sampling rate
    - Converts the signal to a normalized spectrogram
    - Labels it as healthy (0) or faulty (1)

    Args:
        healthy_dir (str): Directory path containing healthy signal files.
        faulty_dir (str): Directory path containing faulty signal files.

    Returns:
        pd.DataFrame: DataFrame containing the following columns:
            - 'filename': Name of the signal file
            - 'label': 0 for healthy, 1 for faulty
            - 'fs': Sampling frequency
            - 'signal': Original 1D signal
```

```
        - 'spectrogram': 2D spectrogram representation
    """


def load_signal(filepath):
    """
    Load a time-domain signal from a text file.

    This function reads a .txt file containing signal values, ignores metadata
    or comment lines that start with '%' and returns the signal as a NumPy array.

    Args:
        filepath (str): Path to the .txt file containing the signal.

    Returns:
        np.ndarray: 1D array of float values representing the signal.
    """


def load_all_signals(healthy_dir, faulty_dir, output_pkl_path):
    """
    Load and label all signal files from healthy and faulty directories.

    This function reads all .txt files in the provided directories,
    assigns label 0 to healthy signals and 1 to faulty ones,
    and stores them in a pandas DataFrame along with filenames.
    Optionally, the resulting DataFrame can be saved as a .pkl file.

    Args:
        healthy_dir (str): Path to directory containing healthy signals.
        faulty_dir (str): Path to directory containing faulty signals.
        output_pkl_path (str): Path to save the resulting DataFrame as a pickle file (.pkl).
                               If None or empty, the file won't be saved.

    Returns:
        pd.DataFrame: DataFrame containing signal data, labels, and filenames.
```

“””

def fix_signal_length(signal, desired_length=25000):
"""
Adjust the length of a 1D signal to a fixed size by trimming or padding.

If the signal is longer than desired_length, it will be center-cropped.
If the signal is shorter, it will be zero-padded at the end.
If it's already the correct length, it is returned unchanged.

Args:
signal (np.ndarray): Input 1D signal array.
desired_length (int): Target length of the signal.

Returns:
np.ndarray: Signal of shape (desired_length,), trimmed or padded as needed.
“””

def normalize_signal(signal):
"""
Normalize a 1D signal using Min-Max scaling to the [0, 1] range.

This function reshapes the input signal to 2D, applies sklearn's MinMaxScaler,
and flattens it back to 1D after scaling.

Args:
signal (np.ndarray): 1D NumPy array representing the raw signal.

Returns:
np.ndarray: Normalized signal with values scaled between 0 and 1.
“””

4.2. Training Script (`src/train.py`):
"""

This script trains the machine learning model using preprocessed data.

Functions:
- create_model(input_length=25000, num_filters=64, dropout_rate=0.3, max_pool=5):Build and return a 1D Convolutional Neural Network for binary classification.
- create_model_2d(input_shape, num_filters=32, dropout_rate=0.5):Build and return a 2D Convolutional Neural Network for binary classification.
- random_search_hyperparameters_2d(X_train, y_train, X_test, y_test):Perform random search to tune hyperparameters for a 2D CNN model.
- random_search_hyperparameters(X_train, y_train, X_test, y_test):Perform random search over a set of hyperparameters for a 1D CNN model.

"""

```
def create_model(input_length=25000, num_filters=64, dropout_rate=0.3, max_pool=5):
    """
    Build and return a 1D Convolutional Neural Network for binary classification.

    The model consists of 3 Conv1D layers with increasing filters and decreasing kernel sizes,
    followed by MaxPooling, Dropout, and GlobalAveragePooling for dimensionality reduction.
    Final classification is done with a Dense layer and softmax activation.

    Args:
        input_length (int): Length of the input 1D signal.
        num_filters (int): Number of filters for the first convolutional layer.
                           The next conv layers use 2x and 4x this value.
        dropout_rate (float): Dropout rate used after the second convolution.
        max_pool (int): Kernel size for the MaxPooling1D layer.

    Returns:
        tf.keras.Model: Compiled Sequential Keras model ready for training.
```

```
    """


def create_model_2d(input_shape, num_filters=32, dropout_rate=0.5):
    """
    Build and return a 2D Convolutional Neural Network for binary classification.

    This model is designed for classifying 2D representations of signals
    (e.g., spectrograms or STFTs). It includes two Conv2D layers with ReLU activation,
    followed by Batch Normalization, MaxPooling, Dropout for regularization,
    and two fully connected layers for final classification.

    Args:
        input_shape (tuple): Shape of the input data (height, width, channels),
                             e.g., (128, 128, 1) for grayscale spectrograms.
        num_filters (int): Number of filters in the convolutional layers.
                           Both Conv2D layers use this value.
        dropout_rate (float): Dropout rate applied before the final Dense layers
                              to reduce overfitting.

    Returns:
        tf.keras.Model: A compiled Keras Sequential model ready for training.

    Example:
        >>> model = create_model_2d((128, 128, 1), num_filters=32, dropout_rate=0.5)
        >>> model.summary()
    """


def random_search_hyperparameters_2d(X_train, y_train, X_test, y_test):
    """
```

```
    Perform random search to tune hyperparameters for a 2D CNN
model.

    This function performs randomized hyperparameter search over a
predefined grid
    to find the best-performing 2D CNN configuration on the given
training and test data.
    For each sampled configuration, the model is trained and evaluated,
    and the best model is saved based on validation accuracy.

    Args:
        X_train (np.ndarray): Training input data of shape (num_samples,
height, width, channels).
        y_train (np.ndarray): Corresponding labels for training data.
        X_test (np.ndarray): Test input data.
        y_test (np.ndarray): Test labels.

    Saves:
        best_2DCNN_model_randomsearch.h5: The best-performing
trained model based on validation accuracy
                                    is saved to the ../models/ directory.

    Prints:
        - Training and validation curves for each configuration
        - Best hyperparameters and final validation accuracy

    Example:
        >>> random_search_hyperparameters_2d(X_train, y_train, X_test,
y_test)
    “""
   pass

def random_search_hyperparameters(X_train, y_train, X_test, y_test):
    """
    Perform random search over a set of hyperparameters for a 1D CNN
model.

    This function searches different combinations of learning rate, batch
size,
```

number of filters, and epochs. It trains a model for each configuration, tracks
validation accuracy, and saves the best-performing model to disk.

Args:
X_train (np.ndarray): Training input signals of shape (num_samples, signal_length).
y_train (np.ndarray): Training labels.
X_test (np.ndarray): Test input signals.
y_test (np.ndarray): Test labels.

Returns:
None. Saves the best model to 'best_1D_model_randomsearch.h5' and prints best params.
“""
pass

4.3. Evaluation Script (`src/evaluate.py`):

"""
This script evaluates the trained model's performance.

Functions:
- classify_signal_1d(filepath, model_path):Classify a raw 1D vibration signal using a trained 1D-CNN model.
- classify_signal_2d(filepath, model_path):Classify a 1D vibration signal as 'Healthy' or 'Faulty' using a trained 2D CNN model.
"""

def classify_signal_1d(filepath, model_path):
"""
Classify a raw 1D vibration signal using a trained 1D-CNN model.

This function loads a signal from a given .txt file, applies MinMax normalization,
reshapes it to match the model's input format, and uses a trained model to predict
whether the signal corresponds to a healthy or faulty device.

Args:
filepath (str): Path to the input .txt file containing the raw signal.

```
        model_path (str): Path to the saved 1D-CNN model (in .h5 format).

    Returns:
        tuple: A tuple (predicted_class, confidence) where:
            - predicted_class (int): 0 for healthy, 1 for faulty.
            - confidence (float): Confidence score (between 0 and 1) of the
prediction.

    Example:
        >>> classify_signal_1d("data/sample_signal.txt", "models/
best_1D_model.h5")
        🧠 Prediction: Healthy (94.28%)
        (0, 0.9428)
    """
    pass

def classify_signal_2d(filepath, model_path):
    """
    Classify a 1D vibration signal as 'Healthy' or 'Faulty' using a trained
2D CNN model.

    This function performs the following steps:
    1. Loads a raw signal from a .txt file (ignoring metadata lines starting
with '%').
    2. Fixes the signal length to a standard size (default 25000).
    3. Converts the fixed signal into a spectrogram (2D representation).
    4. Reshapes the spectrogram to match the input format of the trained
2D CNN model.
    5. Predicts the class using the model and prints the predicted label
and confidence.

    Args:
        filepath (str): Path to the input .txt file containing the signal.
        model_path (str): Path to the saved Keras model (HDF5 format).

    Returns:
        Tuple[int, float]: Predicted class (0 = Healthy, 1 = Faulty), and
confidence score.
    """
    pass
```

5. Model and Architecture Description

Model 1: 1D Convolutional Neural Network

- **Input Shape:** (25000, 1) representing a 1D vibration signal of fixed length
- **Architecture:**
  - Conv1D with num_filters, kernel size 9, stride 2, activation ReLU
  - MaxPooling1D with pool size max_pool and stride 2
  - Conv1D with num_filters * 2, kernel size 7, stride 2, activation ReLU
  - Dropout with rate dropout_rate
  - Conv1D with num_filters * 4, kernel size 5, stride 2, activation ReLU
  - GlobalAveragePooling1D for dimensionality reduction
  - Dense(2) with softmax activation for binary classification
- **Purpose:** Efficient processing of raw time-series sensor data for fault detection
- **Regularization**: Dropout to prevent overfitting

Model 2: 2D Convolutional Neural Network (Spectrogram-based)

- **Input Shape:** (128, 128, 1) representing normalized spectrogram images
- **Architecture:**
  - Conv2D with num_filters, kernel size (3, 3), activation ReLU
  - BatchNormalization
  - MaxPooling2D with pool size (2, 2)
  - Conv2D with num_filters, kernel size (3, 3), activation ReLU
  - MaxPooling2D with pool size (2, 2)
  - Flatten
  - Dropout with rate dropout_rate
  - Dense(64) with activation ReLU
  - Dense(2) with softmax activation for binary classification
- **Purpose:** Transform time-domain data into a frequency-domain representation and use spatial filters to extract robust fault-related features

- **Regularization:** BatchNormalization and Dropout

6. Data Description

- **Source:**
  Vibration signals collected from IoT sensors installed on industrial equipment. The dataset includes both *healthy* and *faulty* samples corresponding to different operating conditions.
- **Format:**
  Each signal is stored in a .txt file with header metadata and a list of signal values sampled over time.
- **Header Metadata (in each file):**
- % Min_X
- % Max_X
- % Unit_Y
- % Unit_X
- % NoOfItems
- **Preprocessing Applied:**
  - **Length Normalization:** All signals are trimmed or padded to 25,000 samples.
  - **Min-Max Normalization:** Each signal is scaled to the range [0, 1].
  - **Spectrogram Transformation:** For the 2D CNN pipeline, signals are converted into log-scaled spectrograms with shape (128, 128).
- **Labels:**
  - 0: Healthy device
  - 1: Faulty device
- **Dataset Size:**
  - **Healthy Samples:** 300

- **Faulty Samples:** 300
- **Total:** 600 signals

7. How to Run

1. Clone this repository and install dependencies:
2. git clone https://github.com/fatememajdi/signal_classification.git
3. cd signal_classification
4. pip install -r requirements.txt
5. Run the notebooks in order:
   - First: notebooks/train_hyperparameter_search.ipynb
   - Then: notebooks/train_hyperparameter_search_2d.ipynb
   - Finally: notebooks/evaluate.ipynb
6. Make sure the models/ directory is created and writeable for saving trained models.
7. Input .txt signal files should be placed under data/raw/Normal Condition/ and data/raw/Looseness/.

8. Outputs

- Trained model saved in `models/` directory as `.h5` file.

9. Maintenance Instructions

- Update dependencies in `requirements.txt` as needed.
- Re-run training whenever significant changes are made to preprocessing or model architecture.
- Periodically review data for drift or inconsistencies.