## CONTENTS

1 Example

2 Contents
   2.1 Installation .......................................................... 5
   2.2 jDAS denoising ......................................................... 6
   2.3 (Re)training jDAS ....................................................... 7
   2.4 Technical details ..................................................... 7
   2.5 Additional resources ............................................... 10
   2.6 License ................................................................. 11
   2.7 Citing jDAS ............................................................ 11
jDAS is a self-supervised Deep Learning model for denoising of Distributed Acoustic Sensing (DAS) data. The principle that underlies jDAS is that spatio-temporally coherent signals can be interpolated, while incoherent noise cannot. Leveraging the framework laid out by Batson & Royer (2019; ICML), jDAS predicts the recordings made at a target channel using the target’s surrounding channels. As such, it is a self-supervised method that does not require “clean” (noise-free) waveforms as labels.

Retraining the model on new data is quick and easy, and will produce an optimal separation between coherent signals and incoherent noise for your specific dataset:

```python
from jDAS import JDAS
jdas = JDAS()
data_loader = jdas.init_dataloader(data)
model = jdas.load_model()
model.fit(data_loader, epochs=50)
```

Denoising your data is then done through a single function call:

```python
clean_data = jdas.denoise(data)
```

That’s all!
The example below is taken from a submarine DAS experiment conducted offshore Greece. At around 25 seconds and earthquake hits the DAS cable and induces a spatio-temporally coherent strain field. jDAS removes the incoherent background noise while keeping the earthquake signals.

Note that some aliasing artifacts have been introduced in rendering this static JPEG. A code example to reproduce this figure is included in the examples directory of the project.
2.1 Installation

2.1.1 Requirements

jDAS depends on the following Python libraries:

- **TensorFlow (>= 2.2.0)**: while training and inference is much faster on a GPU, the CPU version of TensorFlow is sufficient in case problems arise installing the CUDA dependencies.
- **NumPy** and **SciPy** for numerical manipulations.
- **(Optional) Matplotlib** for visualisation.
- **(Optional) h5py** for IO.
- **(Optional) Jupyter notebook or lab** to run the examples

The optional dependencies are required to run the examples. All of these can be installed withAnaconda:

```
conda install -c conda-forge "tensorflow-gpu>=2.2.0" numpy scipy matplotlib h5py notebook
```

Or through PyPI:

```
pip install "tensorflow-gpu>=2.2.0" numpy scipy matplotlib h5py notebook
```

2.1.2 Setting-up jDAS

To obtain the jDAS source code, you can pull it directly from the GitHub repository:

```
git clone https://github.com/martijnende/jDAS.git
```

No additional building is required. To test the installation, try running one of the examples Jupyter notebooks in the examples directory.

Please open a ticket under the tab “Issues” on the GitHub repository if you have trouble setting-up jDAS.
2.2 *jDAS* denoising

Denoising DAS data with *jDAS* is fast and easy, and can be summarised in 3 steps:

1. Instantiate the *jDAS* class.
2. Load a pretrained model with *JDAS.load_model()*.
3. Call the denoising routine *JDAS.denoise()* with the data to be denoised.

**Load a *jDAS* model**

The *jDAS* repository already contains a pretrained model that is loaded by default when calling *JDAS.load_model()* without additional arguments. If a different model is available (e.g. one that is trained on a specific dataset), it can be loaded by pointing *model_path* to the directory where the frozen model is stored (e.g. *JDAS.load_model(model_path="path/to/model/saved-model.h5")*).

**Denoise the DAS data**

The workhorse of *jDAS* is the *denoise()* routine. Under the hood, this routine breaks up the data into small chunks of 2048 time samples and reconstructs the data channel-by-channel, after which the junks are concatenated to obtain the *approximately* same shape as the input; since the input is broken up into an integer number of fixed-size chunks, the output for a single DAS channel will be of size # of chunks x 2048.

In some cases it could happen that parasitic low-frequency artifacts are introduced, which can be filtered out with a standard bandpass filter. This post-processing can be enabled by setting *postfilter=True* (default: False) when calling *denoise()*). The frequency band (and temporal sampling frequency) are specified by the *filter_band* argument. These frequencies are given in Hertz.

Example of denoising the data, including post-process filtering in a 1-10 Hz frequency band (with a sampling frequency of 50 Hz):

```python
jdas = JDAS()
model = jdas.load_model()
clean_data = jdas.denoise(noisy_data, postfilter=True, filter_band=(1, 10, 50))
```

Note that when calling *JDAS.denoise* for the first time in a session, Keras/TensorFlow will rebuild and optimise the model, which takes some time (of the order of 10 seconds). Once this initiation step is done, the subsequent calls to the model are very fast.

**Data requirements**

The DAS data are assumed to be organised in a 2D matrix, for which each row represents one DAS channel (point along the fibre) and each column represents one time sample. So for $N_{ch}$ channels and $N_{t}$ time samples, the data shape is $(N_{ch}, N_{t})$. Moreover, the data are assumed to be Euclidean, meaning that the spacing between each sample is constant in both space and time (constant gauge length and time-sampling frequency).

The pretrained model was trained on a dataset that was bandpass filtered in a 1-10 Hz pass band, and was sampled at 50 Hz with a gauge length (and channel spacing) of 19.2 m. For data with a different gauge length, retraining of the model is required. However, retraining is not necessary if the ratio of the frequency pass band to the Nyquist frequency remains fixed. To give an example, a pass band of 1-10 Hz sampled at 50 Hz is the same as a pass band of 5-50 Hz sampled at 250 Hz. However, since retraining is relatively fast, it is recommended to do so regardless.

A last requirement is that the DAS data be approximately normalised before being passed onto the neural network. This is automatically taken care of by the *denoise()* routine, but it is something to keep in mind when calling the trained model directly.
2.3 (Re)training jDAS

While in some cases the pretrained model provided in the GitHub repository will perform reasonably well out of the box, in practice, retraining the model is recommended, particularly in the following scenarios:

1. A new experiment was conducted (e.g. in a different location, or with different interrogator settings like the gauge length or maximum sensing distance).

2. The conditions during one experiment strongly vary. This can be the case when new noise sources are introduced (construction works on-land, microseismic noise, etc.), or when the signal-to-noise ratio significantly changes.

3. One particular event of interest, such as a major earthquake, occurs. In this case the model can be trained in “single-sample mode”: instead of training on a large data set and optimising the model parameters for a (potentially) wide data range, the training is done on a very specific data range. Consequently, the jDAS model will try to achieve the best denoising performance for this specific data set, at the cost of generalisation.

Note that multiple models can be trained for different conditions (e.g. nighttime/daytime, on-land and submarine segments of the cable, etc.).

The examples/retraining_example.ipynb notebook covers in detail how to prepare the data, retrain the model, and save the model state for later use. The bare-minimum procedure for retraining is as follows:

```python
from jDAS import JDAS
jdas = JDAS()
data_loader = jdas.init_dataloader(data)
model = jdas.load_model()
model.fit(data_loader, epochs=50)
```

Depending on the characteristics of the data, retraining can take anywhere between a minute and an hour on a standard GPU, but in most cases it is expected to take only a few minutes. When the appropriate callbacks are defined (included in the tutorial in the examples directory), the retrained model is saved to a user-defined location, and can be loaded prior to denoising as:

```python
model = jdas.load_model("path/to/saved-model.h5")
clean_data = jdas.denoise(data)
```

2.4 Technical details

This section describes first qualitatively, then quantitatively, the underlying principles of jDAS. These descriptions are an interpretation of the framework laid out by Batson & Royer (2019; ICML), who developed a detailed and thorough body of theory with additional proofs and numerical demonstrations. In this section we will restrict ourselves to the application of $J$-invariance in Deep Learning.

2.4.1 Qualitative description

Consider an example of a checkerboard. If someone were to cover a part of the checkerboard with a piece of paper, you would still be able to predict the pattern that is hidden by the paper with great accuracy. This is because the checkerboard pattern exhibits long-range correlations that can be used for interpolation (and extrapolation). On the other hand, if someone were to introduce a random speckle pattern onto the checkerboard, the details of the speckle pattern underneath the cover paper cannot be predicted; the speckle pattern exhibits no spatial correlations, and an observation of the speckles at one location cannot be used to inform predictions about a different location.

This notion that long-range correlations can be interpolated, while short-range correlations cannot, is what drives jDAS. Imagine that you’d be given a checkerboard sprayed with a black-and-white speckle pattern (for clarity shown in red and
cyan in Fig. 1), but that a few tiles are missing. You are then tasked to reconstruct those tiles as accurately as possible. Aside from reconstructing the missing tiles, you could decide to add a self-made speckle pattern on top. But because you don’t know exactly which speckle goes where, you will likely never guess all the speckles correctly. The best you can do to reconstruct the missing tiles is to estimate the average of the speckles, which is zero if you assume that black and white speckles cancel each other out. Hence, your reconstruction will be informed by the long-range patterns on the checkerboard, but does not include all of the individual speckles. If you now repeat this procedure for different parts of the checkerboard, reconstructing a few tiles at a time, you end up with a reconstruction with no speckle noise.

A similar idea underlies the jDAS filtering approach. Given some DAS data with a spatial and a temporal component, a Deep Learning model can learn to extract correlated patterns in the data, and use those to interpolate gaps in the data. If we create a gap in the data and ask the jDAS model to predict what is inside the gap, and systematically repeat this procedure such that all the data points are “gapped” once, we can collect the Deep Learning predictions for each gap, and put them together to make a noise-free reconstruction of the DAS data. And note that this procedure is entirely based on the presence (or absence) or coherent patterns; we do not need to know a-priori what the noise-free data actually look like. This renders jDAS a so-called “self-supervised” Deep Learning method. The main advantage over “supervised” methods (for which you need to know what the clean data look like) is that you can easily retrain the model on new data (for instance: a new DAS experiment in a different location).

### 2.4.2 Quantitative description

To make the above description more quantitative and precise, define a feature-space partition $J$. In the case of an image, the feature-space is defined by the pixels, so $J$ would represent a patch of pixels. The values of the pixels in $J$ are collectively denoted by $x_J$. Let’s now define some function $f : x \to y$, which takes $x$ as an argument and produces some output $y$. We say that this function is $J$-invariant if $f(x)_J = y_J$ does not depend on $x_J$.

To bring this definition back to the example of the checkerboard, the colour of the tiles (including the speckles) at a given location is denoted by $x$, and we hide a part of the checkerboard under a piece of paper (the partition $J$). We then give $x$ to a function $f$ that produces a reconstruction of the input, $y$. But as we’ve seen above, to make this reconstruction we do not necessarily need to see what is underneath the paper ($x_J$) in order to make a good reconstruction ($y_J$). We can therefore say that interpolating the checkerboard patterns is a $J$-invariant operation.

It would of course be a trivial exercise to predict $y_J$ if we had direct access to $x_J$, which is basically the identity operation. In order to efficiently train a Deep Learning model to not learn the identity operation, we need to restrict
the input of our model to the complement of $J$, denoted by $J^c$. In that way, the Deep Learning model needs to use the surroundings of $x_j$ to predict $y_{J^c}$. Practically this is achieved through a masking operation $\Pi_{J^c}(x)$, which sets all the values of $x$ outside of $J$ to zero.

As opposed to the original procedure adopted by Batson & Royer (2019), we train our Deep Learning model on batches of data, sampled from a larger dataset, and we try to optimise the model parameters by averaging the performance over an entire batch $K$. Let $f(\cdot|\theta)$ denote the Deep Learning model parametrised by $\theta$. The model input for the $k$-th sample ($k \in K$) is then $u_k := \Pi_{J^c_k}(x_k)$, and its output is $v_k := \Pi_{J_k}(f(u_k|\theta))$. The training input is then defined as:

$$\hat{\theta} = \arg\min_{\theta} \frac{1}{|K|} \sum_{k \in K} ||v_k - \Pi_{J_k}(x_k)||^2$$

While this precise training objective is a bit heavy on the notation, it says nothing more but “find the model parameters $\theta$ that minimise the mean squared difference between $x_j$ and the prediction $y_j$ directly”.

### 2.4.3 Model architecture

To describe the jDAS model architecture, we will need to (slightly) move away from the checkerboard analogy, in which the length scale of the correlations was the same along each dimension. In DAS data, however, the two dimensions represent time and space, and the correlations of interest have different wavelengths in each dimension. So instead of applying a square patch like in Fig. 1, we mask one waveform recorded at a random DAS channel by setting it to zero (“blanking”). The blanked DAS channel defines the partition $J$, and so the target for the model is to predict the waveform in $J$ using only the neighbouring DAS channels ($J^c$). In total we use a set of 11 consecutive channels, each 2048 time samples in length. The pretrained model provided in the GitHub repository was trained at a 50 Hz sampling rate, so 2048 samples corresponds to roughly 41 seconds in time.

The Deep Learning model is based on the U-Net architecture (Ronneberger et al., 2015; MICCAI), and features a number of convolutional layers followed by anti-aliasing and resampling layers, as well as the skip connections that are the hallmark of U-Nets. Empirically we found that anti-aliasing before downsampling improves the model performance, possibly because the progressive downsampling brings equivalent Nyquist frequency way below the data frequency band (1-10 Hz). See Zhang (2019; ICML) for a detailed exhibition of internal anti-aliasing.
2.5 Additional resources

Peer-reviewed publication:

```latex
@article{vandenEnde2021,
    author={van den Ende, Martijn Peter Anton and Lior, Itzhak and Ampuero, Jean-Paul and Sladen, Anthony and Ferrari, André and Richard, Cédric},
    title={A Self-Supervised Deep Learning Approach for Blind Denoising and Waveform Coherence Enhancement in Distributed Acoustic Sensing Data},
    publisher={IEEE Transactions on Neural Networks and Learning Systems},
    doi={10.31223/X55K63},
    year={2021},
    volume={0}
}
```

Preprint (content-wise identical to peer-reviewed publication):

```latex
@article{vandenEnde2021_preprint,
    author={van den Ende, Martijn and Lior, Itzhak and Ampuero, Jean-Paul and Sladen, Anthony and Ferrari, André and Richard, Cédric},
    title={A Self-Supervised Deep Learning Approach for Blind Denoising and Waveform Coherence Enhancement in Distributed Acoustic Sensing Data},
    publisher={EarthArxiv},
    doi={10.31223/X55K63},
    year={2021},
    month={Mar}
}
```

Direct link to preprint: EarthArxiv

YouTube explainer video

Tutorials

Tutorials and usage examples are available from the examples directory in the main repository.
2.6 License

MIT License

Copyright (c) 2021 Martijn van den Ende

Permission is hereby granted, free of charge, to any person obtaining a copy of this software and associated documentation files (the "Software"), to deal in the Software without restriction, including without limitation the rights to use, copy, modify, merge, publish, distribute, sublicense, and/or sell copies of the Software, and to permit persons to whom the Software is furnished to do so, subject to the following conditions:

The above copyright notice and this permission notice shall be included in all copies or substantial portions of the Software.

THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.

2.7 Citing jDAS

For use of jDAS in scientific applications, please consider citing the following publication:

@article{vandenEnde2021,
  author={van den Ende, Martijn Peter Anton and Lior, Itzhak and Ampuero, Jean-Paul and Sladen, Anthony and Ferrari, André and Richard, Cédric},
  title={A Self-Supervised Deep Learning Approach for Blind Denoising and Waveform Coherence Enhancement in Distributed Acoustic Sensing Data},
  publisher={EarthArxiv},
  doi={10.31223/X55K63},
  year={2021},
  volume={0}
}