TorchSig: A GNU Radio Block and New Spectrogram Tools for Augmenting ML Training

Phil Vallance PVALLANCE@LTSNET.NET
Erebus Oh EOH@LTSNET.NET
Justin Mullins JMULLINS1@LTSNET.NET
Manbir Gulati MANBIRGULATI@GMAIL.COM
Jared Hoffman JHOFFMAN@APPLIED-INSIGHT.COM
Matt Carrick MATTHEW.CARRICK@PERATONLABS.COM

Abstract

TorchSig uses machine learning (ML) to detect and classify digitized radio frequency (RF) signals. Recent updates and improvements to TorchSig are given, as well as novel features for image-only spectrogram generation and training, reducing memory and computational burdens and making training much faster. A new GNU Radio out-of-tree (OOT) block is provided which uses a TorchSig ML model for detecting signals in real-time.

1. Introduction

TorchSig (TorchSig, a) is an open-source system for machine learning (ML) of digitized radio frequency (RF) signals. TorchSig has two main elements: machine learning models and dataset generation. The system provides for the training of machine learning models with simulated RF data in addition to a variety of pretrained downloadable models. TorchSig also includes a built-in signal generator for generating and simulating RF signals, impairments such as noise, and transforms such as resampling a signal to change its bandwidth. Figure 1 highlights the interconnections between major components within the TorchSig software. This paper gives background information on TorchSig and its software components, a new method for creating and training against synthetic spectrograms, the introduction of a GNU Radio block for plotting labeled spectrograms, and proposes ideas for follow on work.

1.1. Machine Learning Models

TorchSig provides two types of machine learning algorithms, referred to as narrowband and wideband modes. The narrowband mode (Luke Boegner, 2022a) is applied to IQ samples to perform modulation recognition. It is assumed that the input to narrowband mode has already been preprocessed by a channelizer in order to isolate a single signal in frequency, downconvert it to complex baseband and then isolate the energy in time through an energy detector before applying the modulation recognition capability. The narrowband mode is implemented by EfficientNet models (Tan & Le, 2019) and an XCiT models (Alaaeldin El-Nouby, 2021).

The wideband mode (Luke Boegner, 2022b) relaxes some of these constraints by computing a spectrogram or "waterfall" plot and then performing signal detection in time and

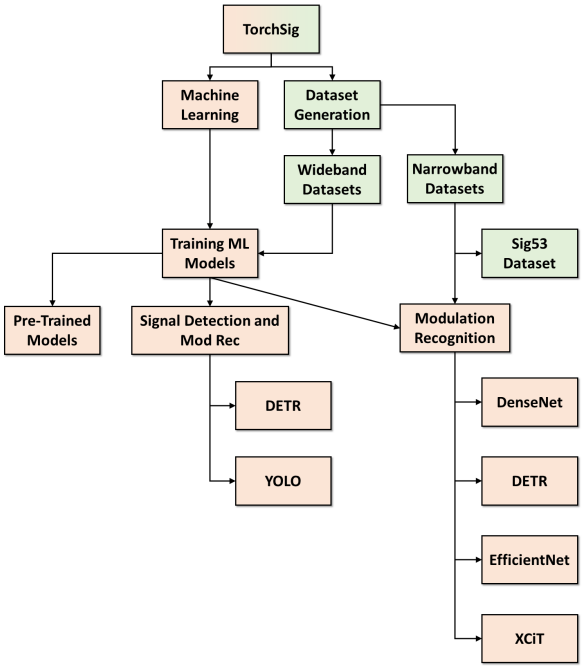

Figure 1. A visual representation of the different components within TorchSig and how they interact with one another.

Proceedings of the 14^{th} GNU Radio Conference, Copyright 2024 by the author(s).

frequency and then applies modulation recognition to each of the detected bursts. Figure 2 displays an example spectrogram from the wideband dataset with a couple signals of varied modulation type, center frequency, bandwidth, and burst duration. Multiple models are available for the wideband mode, including those built on YOLO (Glenn Jocher, 2022). Both narrowband and wideband modes are available in pretrained models on the TorchSig website (TorchSig, b).

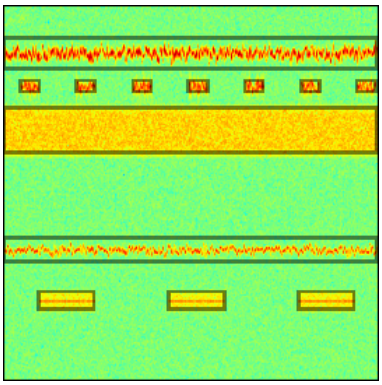

Figure 2. An example spectrogram produced by the TorchSig wideband dataset.

TorchSig utilizes PyTorch compatible datasets, ensuring seamless integration with any training workflow that relies on PyTorch data loaders. Data is loaded from a lightning mapped database (LMDB) (Nic Watson), which is fast enough for real-time training across dozens of GPUs. PyTorch Lightning is used to train the models due to its modular code structure, automated logging, and efficient handling of complex training loops and other features. Convergence is typically reached around 90 epochs or the equivalent of 500 million examples, but a strong model can be trained in under 20 epochs. The narrowband modulations dataset is fast enough that it can be trained on with real-time signal generation, allowing training of much larger models without overfitting.

1.2. Dataset Generation and Sig53

TorchSig has a library of signal modulators to create IQ signals which are then stored in a database to disk. These signals can be used as a benchmarking tool for benchmarking algorithms for algorithmic verification and validation, or performance comparisons between systems. Sig53 is a TorchSig-produced dataset containing 53 different signals from the following (Proakis, 2001):

- Frequency shift keying (FSK)
- Minimum shift keying (MSK)
- Quadrature amplitude modulation (QAM)
- Phase shift keying (PSK)
- Orthogonal frequency division multiplexing (OFDM)
- On-off keying (OOK)
- Pulse amplitude modulation (PAM)

The the majority of the modulated signals, with the exception of OFDM, assume each data symbol is independent and identically distributed (IID) (C. Richard Johnson, 2004) and do not have any explicit framing or protocol structures build into them. On the other hand, the OFDM modulator incorporates some structure into the signal through pilot tones and resource blocks.

The narrowband dataset generates a single signal at complex baseband from the Sig53 set with randomized bandwidth, signal to noise ratio (SNR), and with the application of other transforms and impairments. The wideband dataset also draws from the Sig53 but generates multiple signals with varying center frequencies and bandwidths, also with the application of additional transforms and impairments. More information on the signals, transforms and impairments can be found in (Luke Boegner, 2022a;b).

1.3. Recent Updates

There have been three new releases of TorchSig in 2024: 0.5.1, 0.5.2 and 0.5.2.1 with more planned. These releases have incorporated new features, improved signal processing algorithms, improved the overall speed and fixed bugs. In detail, the releases have changed or added the following:

- Initial release of new image-only dataset tools which create, transform and extract from spectrograms (Section 2)
- 10 times speed improvement through multiprocessing when using more than 32 workers
- Reduced file size for signals stored to disk by using different storage datatype
- Improved randomization by fixing bug that caused some identical signals to be generated
- Tighter bounding boxes for FSK and MSK modulator signals to better isolate those signals in time and frequency
- Reduced the sidelobes in resampling filters from -60 dB to -90 dB
- Improved anti-aliasing filtering to minimize energy wrapping around the $-f_s/2$ and $+f_s/2$ boundary when applying a frequency shift
- Designed a resampling filter to replace the default within SciPy's *resample_poly()* (SciPy) function to reduce aliasing from sample rate change

2. Synthetic Spectrogram Generation and Training

2.1. Spectrograms

A novel feature has been added to TorchSig for creating and training with synthetic spectrograms. A spectrogram is an estimate of the frequency domain as a function of time. Including the time-dimension allows for better visualization for time-varying features in signals, such as signal bursts and frequency hopping. The discrete Fourier transform (DFT) is applied to a sliding time-window of input samples, referred to as the short-time Fourier transform (STFT) (Vaidyanathan, 1993). There are different variants and implementations that build upon the STFT to produce spectrograms, to include trade-offs between time and frequency resolution.

Few-shot object detection on images can be greatly improved by leveraging synthetic data (Lin et al., 2023). The synthetic data includes impairments and augmentations on the original data to develop a larger dataset from a smaller one (Xu et al., 2022). The proposed method uses the Copy-Paste strategy to increase the size of the spectrogram dataset to quickly train a model to identify a signal of interest.

Creating spectrograms can be computationally and memory intensive because of the dependency on the underlying IQ samples which includes a signal modulator, channel effects, transforms and impairments, in addition to the computation and memory of the spectrogram itself. The problem scales quickly as the number of signals in the training dataset increases. Instead, TorchSig uses a method to create synthetic spectrograms where they are built without the creation of the underlying IQ samples which requires less computation and less memory.

Two synthetic spectrograms methods are proposed:

- Recycling and reusing previously generated spectrograms
- Direct creation of synthetic spectrograms

2.2. Recycling and Reusing Spectrograms

One approach to using spectrograms would be to generate unique IQ samples and then compute the corresponding spectrogram. As mentioned, this is computationally and memory intensive. Instead, a method is proposed to recycle and reuse previously generated spectrograms for training a ML model. Rather than generate a large dataset of unique spectrograms, only a handful of example spectrograms are needed in order to represent larger datasets capable of training a YOLO model. Wideband synthetic spectrograms can then be built by assembling a series of smaller narrowband spectrograms. Examples of this usage include TorchSig's *FrequencyHoppingDataset* which overlays narrowband spectrograms into a randomized hopping pattern over a wide bandwidth, *YOLOImageCompositeDataset* which assembles multiple narrowband spectrograms from different protocols into a composite spectrogram, and *CFGSignalProtocolDataset* is a flexible framework for simulating protocol-based waveforms using context free grammar (CFG) (Chomsky, 1956). Figure 3 demonstrates how to use the *FrequencyHoppingDataset*. The code takes a set of narrowband spectrograms and then randomly performs the frequency hops across the wideband spectrum as seen in Figure 4.

```
hopper_ds = YOLOFrequencyHoppingDataset(yolo_bytes_ds,
    [20,80], 10, [100,100], [4,8])
random_hopper_ds = YOLOFrequencyHoppingDataset(yolo_bytes_ds,
    [20,80], 10, [100,100], [4,8], hopping_function=random_hopping)

two_mode_hopper = YOLOCFGSignalProtocolDataset('two_mode_hopping')
two_mode_hopper.add_rule('two_mode_hopping',['12'])
two_mode_hopper.add_rule('two_mode_hopping',['21'])
two_mode_hopper.add_rule('two_mode_hopping',['1'])
two_mode_hopper.add_rule('two_mode_hopping',['2'])
two_mode_hopper.add_rule('12_or_null','12')
two_mode_hopper.add_rule('12_or_null','null', 2)
two_mode_hopper.add_rule('null',None)
two_mode_hopper.add_rule('21_or_null','null', 2)
two_mode_hopper.add_rule('21_or_null','21')
two_mode_hopper.add_rule('12',['1','2'])
two_mode_hopper.add_rule('21',['2','1'])
two_mode_hopper.add_rule('1',hopper_ds)
two_mode_hopper.add_rule('2',random_hopper_ds)
```

Figure 3. The code used to generate a frequency hopping pattern from a set of fixed spectrograms using *FrequencyHoppingDataset*.

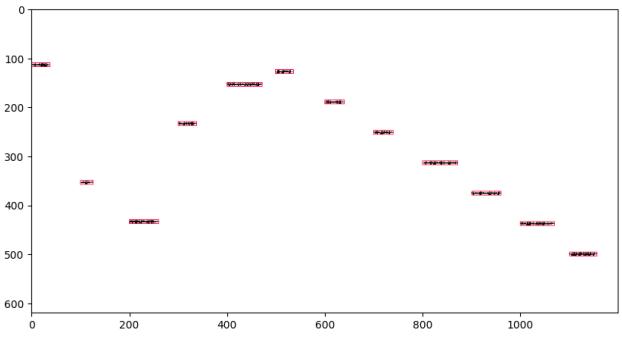

Figure 4. An example of a wideband synthetic spectrogram generated from a set of narrowband spectrograms using *FrequencyHoppingDataset*.

These datasets assemble these wideband spectrograms from components stored as images or implemented as user-defined functions which produce a spectrogram. Images can be narrowband spectrograms, wideband spectrograms with a defined bounding box to designate the signal of in-

terest, drawn through the OpenCV library (OpenCV Team), or any portable network graphic (PNG) image. The user-defined functions can accept parameters to vary the output and must produce a spectrogram. Image transforms will then be applied in order to incorporate real-world effects into the synthetic spectrograms. The transforms include normalization, dynamic range scaling, blur, Gaussian noise and random image rescaling.

2.3. Direct Creation of Synthetic Spectrograms

A method is proposed for the direct creation of synthetic spectrograms and avoiding the creation of IQ samples altogether. Some waveforms have features frequency domain features which can be created directly in the spectrogram, avoiding the need to create the underlying IQ completely. The following proposes creating spectrograms of a chirp-based waveform similar to LoRa (Knight & Seeber, 2016). Figure 5 demonstrates how a synthetic spectrogram for a chirp-based waveform can be created directly without having to create any underlying IQ information using the *CFGSignalProtocolDataset* function.

```
rising_chirp = GeneratorFunctionDataset(
    chirp_generator_function(1, 20, 4, random_height_scale = [0.9,1.1], random_width_scale = [1,2]),
    transforms=add_falling_edge)
falling_chirp = GeneratorFunctionDataset(
    chirp_generator_function(1, 20, 4, random_height_scale = [0.9,1.1], random_width_scale = [1,2]),
    transforms=[add_falling_edge, lambda x: x.flip(-1)])

chirp_stream_ds = CFGSignalProtocolDataset('cfg_signal')
chirp_stream_ds.add_rule('cfg_signal', ['rising_or_falling_stream'] + ['rising_falling_or_null']*12)
chirp_stream_ds.add_rule('rising_falling_or_null', 'rising_or_falling_stream', 1)
chirp_stream_ds.add_rule('rising_falling_or_null', 'null', 1)
chirp_stream_ds.add_rule('null', None)
chirp_stream_ds.add_rule('rising_or_falling_stream', 'rising_stream')
chirp_stream_ds.add_rule('rising_or_falling_stream', 'falling_stream')
chirp_stream_ds.add_rule('rising_stream', ['rising_segment'] + ['rising_segment_or_null']*2)
chirp_stream_ds.add_rule('rising_segment_or_null', 'rising_segment')
chirp_stream_ds.add_rule('rising_segment_or_null', 'null')
chirp_stream_ds.add_rule('rising_segment', ['rising_chirp']*3)
chirp_stream_ds.add_rule('rising_chirp', rising_chirp)
chirp_stream_ds.add_rule('falling_stream', ['falling_segment'] + ['falling_segment_or_null']*2)
chirp_stream_ds.add_rule('falling_segment_or_null', 'falling_segment')
chirp_stream_ds.add_rule('falling_segment_or_null', 'null')
chirp_stream_ds.add_rule('falling_segment', ['falling_chirp']*3)
chirp_stream_ds.add_rule('falling_chirp', falling_chirp)

yolo_chirp_stream_ds = YOLODatasetAdapter(chirp_stream_ds, class_id=0)
```

Figure 5. The code for generating a synthetic spectrogram for a chirp-based waveform is done through the definition of rules on the ordering and selection of symbols using *CFGSignalProtocolDataset*.

From the example,

```
chirp_stream_ds.add_rule('cfg_signal',
  ['rising_or_falling_stream'] +
  ['rising_or_falling_or_null']*12)
```

defines the spectrogram to have one initial data symbol which can be a rising or falling stream, and then up to 12 additional data symbols. The first data symbol must be a rising or falling chirp, and then the next 12 can be rising chirp, a falling chirp, or null which is no data symbol at all. The rules are chained, requiring the *rising_or_falling_stream* variable to be defined as a *rising_stream* or *falling_stream* with equal probability,

```
chirp_stream_ds.add_rule(
  'rising_or_falling_stream',
  'rising_stream')

chirp_stream_ds.add_rule(
  'rising_or_falling_stream',
  'falling_stream')
```

A rising stream is defined as between one and three rising segments,

```
chirp_stream_ds.add_rule(
  'rising_stream', ['rising_segment']
  + ['rising_segment_or_null']*2)
```

where a rising segment is defined as three rising chirps,

```
chirp_stream_ds.add_rule(
  'rising_segment',['rising_chirp']*3)
```

Figure 6 gives a synthetic spectrogram that is created using the reference code.

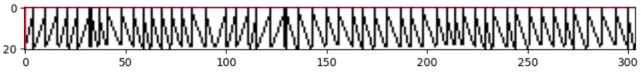

Figure 6. A synthetic spectrogram from a chirp-based waveform created using *CFGSignalProtocolDataset*.

Multiple datasets can be combined into a composite spectrogram to create additional variability and complexity for the ML model to train against. For example, Figure 7 displays a spectrogram created by *FrequencyHoppingDataset* and *CFGSignalProtocolDataset* and adds transforms to simulate noise and dynamic range scaling.

3. GNU Radio Blocks for ML Inference and Display

The out-of-tree (OOT) module *gr-spectrumdetect* incorporates a YOLOv8x TorchSig ML model to detect signals in real time. Figure 8 displays a test flowgraph that is provided with *gr-spectrumdetect* for demonstrating how to apply the *specDetect* block to detect and label signals in a spectrogram, and then plot the result using the *spectrumPlot* block. Figure 9 shows the properties for the *specDetect* block. It is important to note that parameters need to match what the pretrained model is expecting. The block parameters are shown in Figure 9.

From Figure 8, the IQ samples are received from the *UHD USRP Source* block and then transformed into a vector.

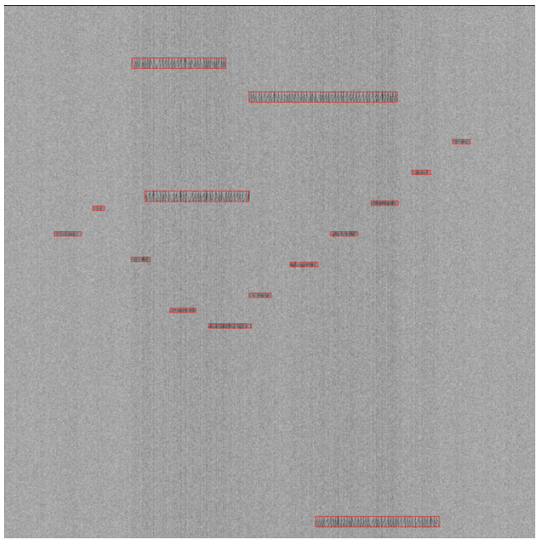

Figure 7. A synthetic spectrogram created through the *FrequencyHoppingDataset* and *CFGSignalProtocolDataset* functions for simulating a frequency hopping waveform and a chirp-based waveform.

The provided ML model operates on a spectrogram with a DFT size of 1024 bins and 1024 total DFTs, for a total number of points of $1024 \cdot 1024 = 1048576$. Therefore the *Stream to Vector* block transforms the serial input into a vectorized output with 1048576 samples. Internally, *specDetect* computes a 1024x1024 spectrogram using Torchaudio and then applies the wideband ML model to locate signals within the spectrogram and label them using bounding boxes. The spectrogram and bounding boxes are then mapped into a dictionary of polymorphic types (PMT) and sent as a message, intended for plotting by the *spectrumPlot* block. The *spectrumPlot* block receives the dictionary of PMTs, displays the spectrogram in a Matplotlib window and then overlays the bounding boxes to highlight the detections. Figure 10 demonstrates an example of inference from the YOLOv8x model within the 2.4 GHz industrial scientific and medical (ISM) band.

Example bash and python scripts are provided within the *gr-spectrumdetect* source code to demonstrate the process of building the provided trained model *detect.pt*. The model is trained over the Sig53 wideband dataset using the same parameters as in the example flowgraph. The *generate.sh* script generates the wideband dataset and stores it as an LMDB. The bash script *make_yolo.sh* converts the TorchSig LMDB into $1024x1024$ YOLO images and text file labels format. The bash script *train.sh* uses the Ultralytics YOLOv8 command line to train a YOLOv8x model for one epoch on the spectrograms. The training starts from the Ultralytics provided pretrained YOLOv8x model on the COCO dataset. The first layer of the YOLOv8x model is frozen while trained with a batch size of 32 and a learning rate of 0.0033329. The weights have a mean average precision 50-95 (mAP50-95) of 89.3% when tested against the impaired wideband Sig53 dataset. Single class mode is used so the model only learns to detect energy. These settings were found to work on ISM band data through hyperparameter experimentation.

4. Future Work

New features are under development and more are being planned for TorchSig, including new signal types, additional features for synthetic spectrogram generation and modification, and tools for incorporating custom data sets. Support will be developed for HuggingFace and PyTorch compatible checkpoints for a variety of model sizes and use cases, facilitating broader applicability and accessibility of our models. Additional transforms are being developed for directly texturing and modifying wideband spectrograms to avoid the computation and generation of IQ samples. Analog modulations amplitude modulation (AM) and frequency modulation (FM) and their variants are being developed for inclusion into the signal modulator library. Label Studio is being integrated which would allow for the simulated dataset to be supplemented with real world data.

References

Alaaeldin El-Nouby, et al. XCiT: Cross-covariance image transformers, 2021. URL https://arxiv.org/abs/2106.09681.

C. Richard Johnson, et al. *Telecommunication Breakdown*. Pearson Prentice Hall, Upper Saddle River, NJ, 2004.

Chomsky, Noam. Three models for the description of language. *IRE Transactions on Information Theory*, 2(3): 113–124, 1956. doi: 10.1109/TIT.1956.1056813.

Glenn Jocher, et al. ultralytics/yolov5: v6.1 - TensorRT, TensorFlow Edge TPU and OpenVINO Export and Inference, February 2022. URL https://doi.org/10.5281/zenodo.6222936.

Knight, Matthew and Seeber, Balint. Decoding lora: Realizing a modern lpwan with sdr. *Proceedings of the GNU Radio Conference*, 1(1), 2016. URL https://pubs.gnuradio.org/index.php/grcon/article/view/8.

Lin, Shaobo, Wang, Kun, Zeng, Xingyu, and Zhao, Rui. Explore the power of synthetic data on few-shot object detection, 2023. URL https://arxiv.org/abs/2303.13221.

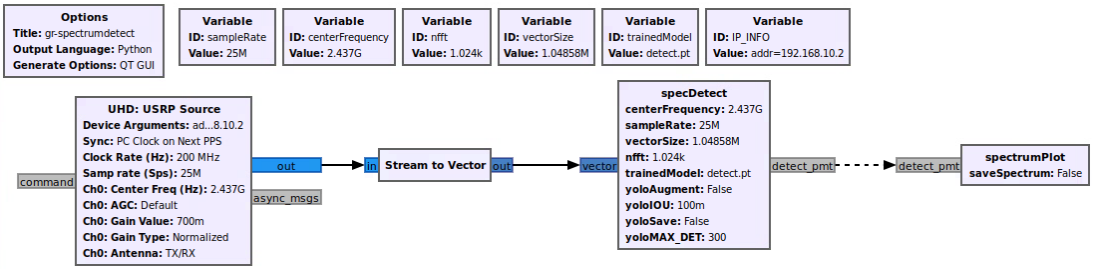

Figure 8. A flowgraph provided with *gr-spectrumdetect* which receives IQ samples from the USRP, turns them into a vector, applies ML inference and plots the resulting labeled spectrogram.

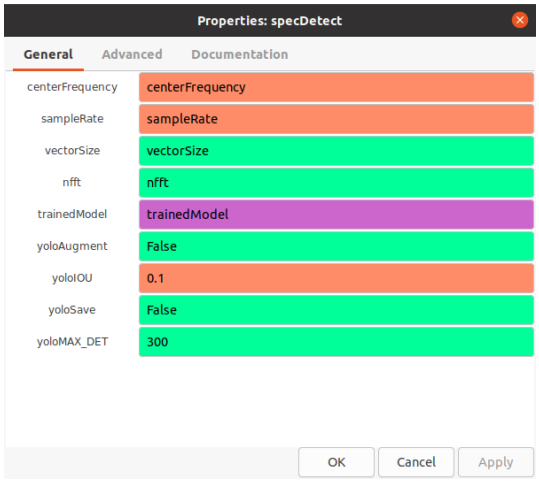

Figure 9. The properties that configure the *specDetect* block.

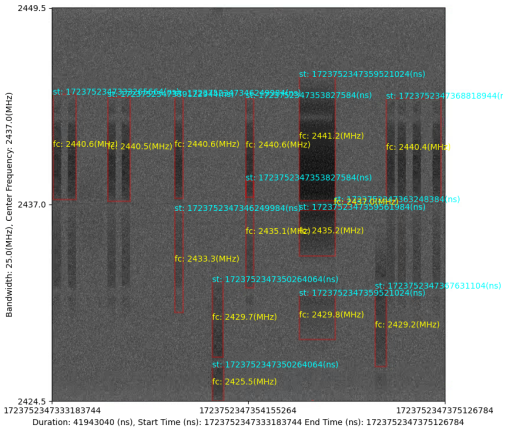

Figure 10. An example of ML inference running on live signals in the ISM Band.

Luke Boegner, et al. Large scale radio frequency signal classification, 2022a. URL https://arxiv.org/abs/2207.09918.

Luke Boegner, et al. Large scale radio frequency wideband signal detection & recognition, 2022b. URL https://arxiv.org/abs/2211.10335.

Nic Watson, et al. py-lmdb. URL https://github.com/jnwatson/py-lmdb/.

OpenCV Team. OpenCV. https://opencv.org/.

Proakis, John G. *Digital Communications, Fourth Edition*. McGraw-Hill, New York, NY, 2001.

SciPy. scipy.signal.resample_poly(). URL https://docs.scipy.org/doc/scipy/reference/generated/scipy.signal.resample_poly.html.

Tan, Mingxing and Le, Quoc. EfficientNet: Rethinking model scaling for convolutional neural networks. In Chaudhuri, Kamalika and Salakhutdinov, Ruslan (eds.), *Proceedings of the 36th International Conference on Machine Learning*, volume 97 of *Proceedings of Machine Learning Research*, pp. 6105–6114. PMLR, 09–15 Jun 2019. URL https://proceedings.mlr.press/v97/tan19a.html.

TorchSig. TorchSig. https://github.com/torchdsp/torchsig, a.

TorchSig. TorchSig Downloads: Code, Data, Models, Papers. https://torchsig.com/dist/downloads.html, b.

Vaidyanathan, P. P. *Multirate Systems and Filter Banks*. Prentice Hall, Upper Saddle River, NJ, 1993.

Xu, Yang, Huang, Bohao, Luo, Xiong, Bradbury, Kyle, and Malof, Jordan M. Simpl: Generating synthetic overhead imagery to address custom zero-shot and few-shot detection problems. *IEEE Journal of Selected Topics in Applied Earth Observations and Remote Sensing*, 15:4386–4396, 2022. doi: 10.1109/JSTARS.2022.3172243.