TorchSig 2.0: Dataset Customization, New Transforms and Future Plans

Erebus Oh EOH@LTSNET.NET
Justin Mullins JMULLINS1@LTSNET.NET
Matt Carrick MATTHEW.CARRICK@PERATONLABS.COM
Matt Vondal MATTHEW.VONDAL@PERATONLABS.COM
Jared Hoffman JHOFFMAN@APPLIED-INSIGHT.COM
Frank Leonardo FRANCIS.LEONARDO@PERATONLABS.COM
Paul Toliver PTOLIVER@PERATONLABS.COM
Rob Miller RMILLER@PERATONLABS.COM

Abstract

TorchSig's v2.0 release improves the training and testing of machine learning (ML) models with synthetic datasets by providing transforms that model important real-world effects like phase noise, quantization, or amplifier nonlinearities. Users also have more dataset generation controls and customization options, such as parameterizing the range of center frequencies, bandwidths, and burst durations. Additionally, users can bring their own data and use TorchSig's framework for data augmentation and model training.

1. Introduction

TorchSig is a free, open-source (TorchSig GitHub) radio frequency (RF) machine learning (ML) software toolkit (TorchSig Website). The software was originally published in 2022, producing what is now referred to as *narrowband* datasets (Boegner et al., 2022a), with a large update later in the year producing the *wideband* datasets (Boegner et al., 2022b). Since it's inception, there have been 18 releases of the TorchSig software. The software creates randomized datasets of synthetic RF signals of 57 analog and digital modulation variants from the modulation families of quadrature amplitude modulation (QAM), orthogonal frequency division multiplexing (OFDM), frequency shift keying (FSK), frequency modulation (FM), and others. The datasets natively use in-phase and quadrature (IQ) complex samples, but can be transformed into other formats such as spectrograms. Datasets can be created on the fly or written to disk using Zarr (Zarr) for post-processing.

Transforms modify datasets to apply desired data characteristics and improve machine learning model generalization. Some transforms simulate real-world, non-linear hardware and channel effects (referred to as *impairments*) such as amplifier compression, phase noise, or intermodulation products. Other transforms implement operations which may not have a real-world equivalent, such as time-reversal or adding the signal derivative, yet which are useful in training the machine learning model.

Datasets can be generated for training and validation or as a flexible signal generator for other purposes. TorchSig has a PyTorch-based processing pipeline for reading in datasets and training ML models with the appropriate IQ samples or spectrograms and their associated metadata. The ML models can be trained for detection, modulation recognition or other applications.

Examples of how to train a custom ML model and then validate the performance are described in Jupyter notebooks within the *examples/* directory. There are a total of 21 Jupyter notebooks (Jupyter) that act as short-form tutorials; five that demonstrate how to create datasets and train ML models and 16 which describe the operations and effects of selected transforms.

TorchSig also provides pretrained ML models and a GNU Radio block *gr-spectrumdetect* (Vallance et al., 2024) which uses selectable YOLO (Ultralytics) models to detect and locate energy within a spectrogram and then perform modulation recognition with a XCiT model. The pretrained models can be downloaded with the *gr-spectrumdetect/examples/trained_models_download.sh* script. The models can be integrated directly or used within the *gr-spectrumdetect* block. The models are:

- 11s.pt: A YOLOv11 size S model trained for energy detection on wideband spectrograms
- detect.pt: A YOLOv8 size X model trained for energy detection on wideband spectrograms

Proceedings of the 15th GNU Radio Conference, Copyright 2025 by the author(s).

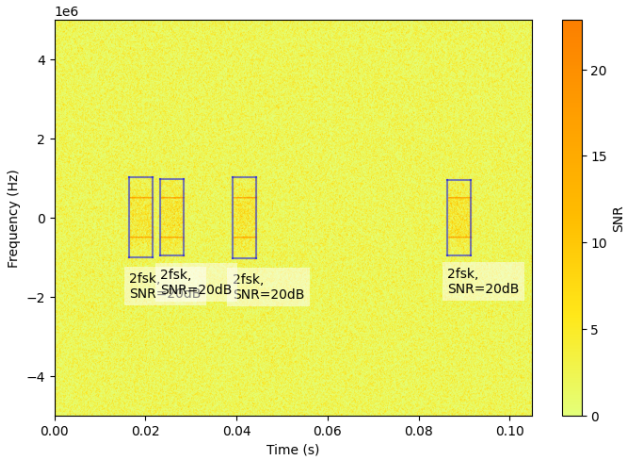


Figure 1. A customized dataset of 2-FSK with fixed SNR, center frequency, bandwidth and burst duration.

- xcit.ckpt: An XCiT model trained for narrowband modulation recognition on IQ data

2. Recent Releases and Updates

TorchSig 1.0, released in March 2025, is a ground-up rewrite from earlier versions to support future growth and new features in follow-on versions. The rewrite maintained the same core functionality and workflows for dataset generation and training models, but with significant changes to the implementation. One of the largest changes is the definition of *dataset metadata* in which users can now customize parameters such as the number of signals to generate, signal to noise ratio (SNR) values, and the burst durations.

TorchSig 1.1, released in April 2025, allows the user further customization of datasets and overall speed improvements. The user is now able to define a range for signal center frequencies and bandwidths (Figure 1). Additionally, all parameters are given in real-world units such as seconds and Hertz (Hz). Significant speed improvements were added by pre-calculating resampling filter weights once and then loading from file as needed and having Zarr write in batches of files as they are processed. Users are also informed of their estimated dataset size and the amount of space remaining on disk.

TorchSig 2.0, released in Summer 2025, includes a new set of RF hardware-based transforms and a unification of the narrowband and wideband datasets. A detailed explanation is given in Section 3.

3. TorchSig 2.0

TorchSig 2.0 builds the foundation for more realistic signals and data. This release includes the following significant changes: the addition of a new set of transforms that mimic real-world hardware impairments, unifying narrowband and wideband datasets, using iterable datasets, a controllable amount of co-channel interference, and a Bring-Your-Own-Data (BYOD) system where users can use TorchSig with their own data.

3.1. New Transforms Simulating RF Hardware Effects

Transforms in TorchSig are used to apply an effect to the training data in order to produce a better performing ML model. In this case, new transforms are added which simulate non-linear effects and other impairments from analog RF hardware. The RF hardware itself is not being simulated, instead it is the non-linear effects of the various components in both the transmit and receive paths which are being simulated and applied.

The transforms used to simulate analog RF hardware are listed below:

Carrier Phase Offset	Carrier Phase Noise
Carrier Frequency Drift	Coarse Gain Change
Digital AGC	Intermodulation Products
IQ Imbalance	Passband Ripple
Nonlinear Amplifier	Quantize
Sampling Clock Drift	Sampling Clock Jitter
Spectral Inversion	Spurs

The *Builder()* class creates IQ signals that are idealized and textbook-like, meaning they are created synthetically and do not contain any hardware imperfections that real-world signals do. Transforms have been added to apply impairments such as non-linear amplifiers and quantization effects to make the simulated signals more similar to those found in the real-world. The intent is that modeling both transmitter and receiver hardware effects in the training data will make it more likely that the ML model will be able to correctly detect and recognize real-world signals based on the synthetic dataset.

3.1.1. Default Parameters and Customization

The transforms are designed with default ranges for the randomized parameters that implement effects seen in low-, mid- and high-quality analog RF hardware. However, their operation can be customized by overwriting the default parameter sets in *impairments.py*.

The parameters are often a range of values as in the case of the quantization transform, having a minimum and maximum number of quantization bits to simulate. Some transforms have parameters which may be selected from a dis-

crete list, such as with intermodulation products which can be selected as 3^{rd} or 5^{th} order. Others such as the spectral inversion transform may have no customizable parameters at all.

Each of the transforms are applied with a given probability. This is designed to mimic real-world situations in which not every non-linearity will impact the signal every time from every transmitter or receiver. For example, not every receiver will have IQ imbalance, therefore the transform is enabled with 50% probability such that the ML model can learn to detect and recognize signals when the impairment is and is not present. Additionally, the probability helps with model generalizability. The likelihood that a transform is being applied can be customized by changing the probability within the *RandomApply()* function.

The following sections describe the various impairment-based transforms.

3.1.2. Passband Ripple

The passband ripple transform simulates the effects from analog filters at the transmitter or receiver. A short 2 or 3 length finite impulse response (FIR) filter is applied to simulate this analog filter effect. The amount of ripple is customizable, but defaults to a maximum of 2 dB.

3.1.3. Carrier Phase Offset

The carrier phase offset transform applies a random phase offset by multiplying with a complex exponential $e^{j\phi}$ where ϕ is of the range $[0, 2\pi)$. The random phase offset represents a time delay and cumulative static phase shift through both transmitter hardware, the channel and receiver hardware before the signal is sampled at the receiver.

3.1.4. Carrier Phase Noise

The carrier phase noise transform represents uncertainty and randomness of the phase of the carrier. Where the phase offset represents a constant phase change, the phase noise θ is a zero-mean Gaussian random variable $\theta \sim \mathcal{N}(0, \sigma_\theta^2)$. The variance σ_θ^2 is defined according to a parts-per-million (PPM) measure, such that for a 10 ppm phase noise error $\sigma_\theta = 10^{-6}$. The transform is implemented by the multiplication of a complex exponential of the form $e^{j\theta}$. Figure 2 demonstrates the effect of phase noise on a quadrature-phase shift keying (QPSK) signal, which appears as a blurring of the constellation points.

3.1.5. Carrier Frequency Drift

The carrier frequency drift transform further represents the randomness and uncertainty of a carrier wave by representing the frequency as a Gaussian random variable. The transform is implemented by the multiplication of a complex exponential of the form $e^{j2\pi(f_{ideal}/f_s)n}$ where f is a Gaussian random variable $f \sim \mathcal{N}(f_{ideal}, \sigma_f^2)$. where f_{ideal} is the perfect or idealized center frequency, f_s is the sampling rate and n is the discrete time index. The variance σ_f^2 is defined in PPM, such that for a 10 ppm frequency error $\sigma_f = 10^{-6}$.

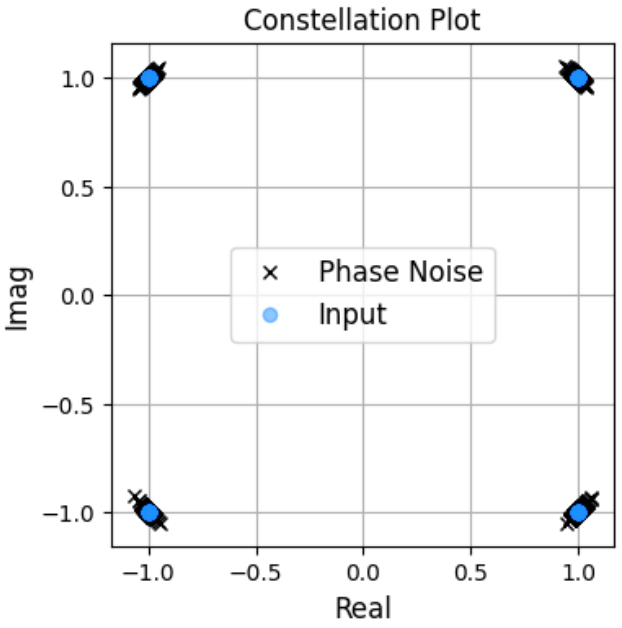


Figure 2. An IQ plot demonstrating the carrier phase noise transform, where the constellation points have some noise and smearing associated with them.

3.1.6. Intermodulation Products

The intermodulation transform models Inter-Modulation Distortion (IMD) products by applying a memoryless nonlinear response to a baseband signal. The user specifies coefficient weights $c[n]$ such that $y[n] = \sum_k c[k-1] * x[n]^k$ is the transformed output. If a coefficient is zero that order product is not computed, enabling a focus only on the desired IMD components of interest, such as third- and fifth-order products.

3.1.7. IQ Imbalance

The I/Q Imbalance transform applies configurable mismatches in amplitude, phase, and DC level between the in-phase (I) and quadrature (Q) components of a signal. These effects are often encountered in direct conversion receivers where they negatively impact performance.

3.1.8. Coarse Gain Change

The coarse gain change represents an immediate change in gain at the receiver. For a real-world receiver, a change in gain would either raise or lower the relative power level for signals while leaving the noise mostly unaffected. However, as TorchSig does not fully model RF receiver hardware, the gain change will impact all aspects of the receive signal equally.

3.1.9. Digital AGC

The digital automatic gain control (AGC) transform simulates an automatic gain control that would be implemented digitally in order to adaptively set the nominal amplitude level. The digital AGC has control logic and feedback that will adapt the gain level to either increase or decrease the incoming signal according to the desired reference level, creating transient effects due to the changing amplitude level.

3.1.10. Nonlinear Amplifier

Two different nonlinear amplifier models are provided. The nonlinear amplifier function-based transform models memoryless amplitude and phase modulations using hyperbolic tangent responses. The hyperbolic response models both a linear region and a non-linear compressive region which causes distortion in the signal. Figure 3 is an example of the nonlinear amplifier effects on a signal. A second transform is available which allows the user to specify a custom table of power and phase values for interpolation as an arbitrary response.

3.1.11. Quantize

The quantize transform implements the quantization function of an analog to digital converter (ADC). The transform takes into account the number of bits in the ADC and a relative power level that simulates if the ADC is receiving the signal at full scale, above full scale or below full scale. ADCs are evaluated by their quantization accuracy, referred to as the effective number of bits (ENOB) (Lyons, 2011). When the signal is at full scale, meaning the input signal amplitude reaches all of the quantization levels of the ADC, then the ENOB will approach the number of bits used in the quantizer. However, when the input signal amplitude is too large the ADC will saturate causing distortion which will reduce the ENOB dramatically. Similarly, then the input signal amplitude is too low not all quantization levels will be activated and therefore the ADC is operating at a lower ENOB.

Figure 4 gives an example of the quantizer transform with 8 bits. The simulated saturation effect drops the ENOB to 1.7 bits, and when the amplitude level is 35 dB too small it reduces the ENOB to 2.4 bits.

3.1.12. Sampling Clock Jitter

The sampling clock jitter impairment simulates uncertainty and error in the sampling clock phase. The impairment simulates the inability for the sampling clock to be at the same relative phase from sample to sample. The error is modeled by adding a zero-mean Gaussian random variable to the commutator arm within a polyphase filter bank-based interpolator (fredric j. harris, 2021). The jitter either increases or decreases the sampling phase, represented by the commutator arm, by a fraction of a sample.

3.1.13. Sampling Clock Drift

The sampling clock drift impairment simulates the error in the sampling clock rate. The sampling rate in real-world systems can and will change over time by a small values. The sampling rate error comes from sources such as manufacturing tolerances and temperature changes. The impairment is simulated by accumulating a zero-mean Gaussian random variable over time and controlling the commutator arm within a polyphase filter bank-based interpolator. The sampling rate is given a nominal value and then either increased or decreased according to the accumulated error over time, a form of a random walk process. The sampling clock drift is a second source of clock error is treated independently of the sampling clock jitter impairment.

3.1.14. Spectral Inversion

Real-valued signals have fully mirrored spectral information in the frequency domain across $f = 0$, referred to as *images*. Either of the images may be used, which is determined by how either the transmitter or receiver hardware is implemented. The transmitted signal $x[n]$ is real-valued, therefore the lower and upper image have the symmetry $\mathcal{F}\{x[n]\} = X\left(e^{j\omega}\right) = X^*\left(e^{-j\omega}\right)$ (Oppenheim & Schaefer, 1999). When the transmit filter selects for the lower frequency image there will be spectral inversion, represented by $x_{lower}[n] = X^*\left(e^{-j\omega}\right)$. The spectral inversion comes from the negative frequency term $-\omega$. Using the property $\mathcal{F}\{y^*[n]\} = Y^*\left(e^{-j\omega}\right)$ (Oppenheim & Schaefer, 1999), the conjugation of the time-domain signal undoes the spectral inversion effect according to $\mathcal{F}\left\{x^*_{lower}[n]\right\} = X\left(e^{j\omega}\right)$.

3.1.15. Spurs

Spurs are the product of non-linearities in both transmitter and receiver hardware. The spurs transform creates a randomized number of spurs in various locations across the full spectrum. The power levels are randomized for each instance, but are held constant across the length of the IQ array. Each spur is implemented as complex exponential

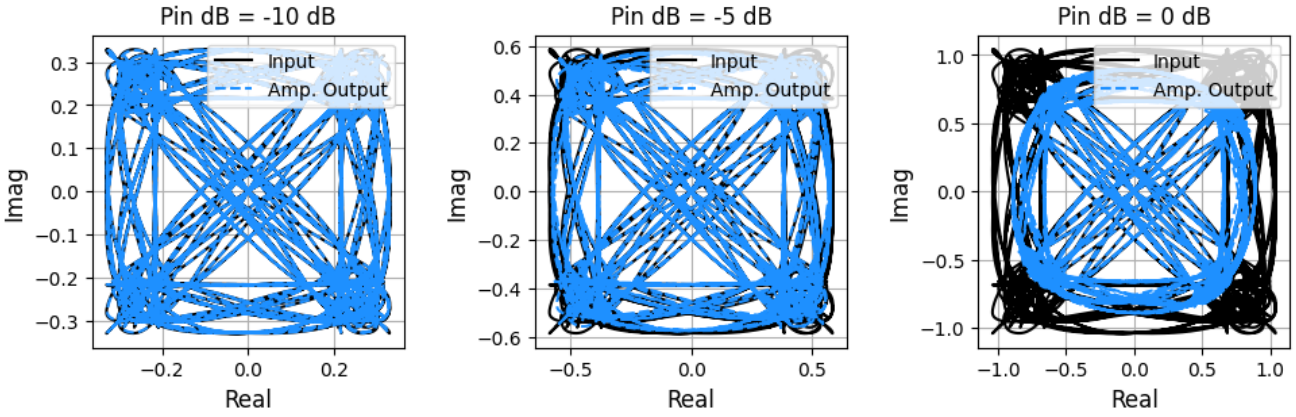


Figure 3. An IQ plot demonstrating the nonlinear amplifier transform for different input power levels and the corresponding compressive effect.

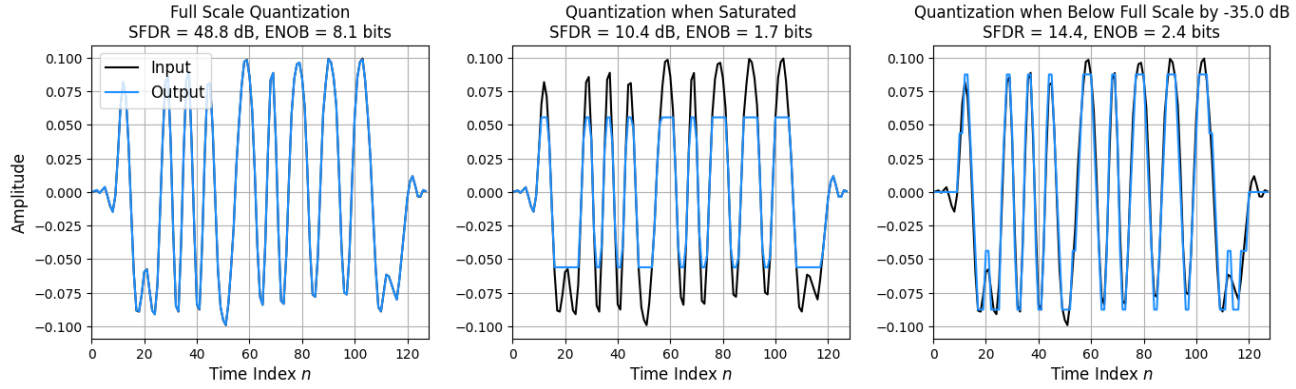


Figure 4. A time domain plot demonstrating the quantize transform when the signal is at full scale, saturated and below full scale.

(unmodulated carrier).

3.1.16. Transmit and Receive Impairment Ordering

Impairment-based transforms are grouped into two lists: one for transmit and one for receive. The transforms within the transmit and receive lists are internally ordered differently such that they accurate represent the processing chains for their respective hardware. For example, the quantize transform is first in the list of transmit impairments, but it is second to last in the receive list. The transmit list of impairments is applied to each singular signal in an isolated fashion, mimicking the transmit hardware. However, the receive list of impairments is applied to the full received signal, which is the summation of all transmitted signals and thermal noise.

The transmit impairments are applied in the following order with a given probability:

- Quantize, 75%
- Sampling Clock Drift, 75%
- Sampling Clock Jitter, 75%
- Passband Ripple, 75%
- IQ Imbalance, 25%
- Carrier Phase Noise, 75%
- Carrier Frequency Drift, 75%
- Carrier Phase Offset, 100%
- Intermodulation Products, 50%
- Nonlinear Amplifier, 75%
- Spurs, 75%
- Spectral Inversion, 25%

The following is the list of receive impairments:

- Intermodulation Products, 50%
- Nonlinear Amplifier, 75%
- Coarse Gain Change, 25%
- Spurs, 75%
- IQ Imbalance, 50%
- Carrier Phase Noise, 75%
- Carrier Frequency Drift, 75%
- Carrier Phase Offset, 100%
- Passband Ripple, 75%
- Sampling Clock Drift, 75%
- Sampling Clock Jitter, 75%
- Quantize, 75%
- Digital AGC, 25%

3.2. Narrowband and Wideband Dataset Unification

In previous versions of TorchSig there was a distinction between "narrowband" and "wideband" datasets. Narrowband was represented by a single signal with a small amount of oversampling and whose burst duration took up a large portion of the total time-series duration. The narrowband dataset was used to train for modulation recognition. Wideband consisted of multiple signals whose burst durations and bandwidths were small in proportion to the area of the spectrogram. The wideband dataset was used to to train for energy detection, finding the signal in both time and frequency.

The distinction between the two datasets has been removed in TorchSig 2.0. Rather than using *NarrowbandMetadata(), WidebandMetadata(), NewNarrowband() or NewWideband()*, all datasets are created and customizable through *DatasetMetadata()* and *TorchSigIterableDataset()*. Note that the old "narrowband" and "wideband" datasets can be easily recreated with the new structure.

The user now has more flexibility and customization when creating datasets with the ability to specify the number of signals and their center frequencies, bandwidths, and burst durations. An example of the customizations are given in Listing 1.

```
dataset_metadata = DatasetMetadata(
    sample_rate = 10e6, # Hz
    fft_size = 1024,
    num_iq_samples_dataset = 1024**2,
    sample_rate = 10e6, # Hz
    num_signals_min = 1,
    num_signals_max = 5,
    snr_db_min = 0,
    snr_db_max = 40,
    signal_duration_min = 10e-3, # seconds
    signal_duration_max = 100e-3, # seconds
    signal_bandwidth_min = 100e3, # Hz
    signal_bandwidth_max = 1e6, # Hz
    signal_center_freq_min = -4e6, # Hz
    signal_center_freq_max = 4e6 # Hz
)
```

Listing 1. Example dataset metadata used to customize datasets.

The default sample rate is *sample_rate = 10e6* (10 MHz) and the default number of bins in the FFT is *fft_size = 1024*. Both of these parameters can be overridden.

The dataset metadata defines each IQ array within the dataset to have a length of $1024^2 = 1048576$ IQ samples. This, alongside the *sample_rate* and *fft_size* parameters, determine the minimum and maximum ranges for the center frequency, bandwidth and burst durations.

Thus, ranges for parameters can be calculated. For the default parameters, the maximum center frequency of a signal is *sample_rate/2* = 5 MHz, with the minimum being *-sample_rate/2* = -5 MHz. The maximum bandwidth is the sample rate, 10 MHz, while the minimum bandwidth is defined by *sample_rate/fft_size* $\approx$ 9.7 kHz. The minimum bandwidth is dictated by the FFT bin width, which also represents the bandwidth of an unmodulated carrier tone. The maximum burst duration is defined by *num_iq_samples_dataset/sample_rate* $= 1048576/10^6 \approx 0.1$ seconds.

The minimum and maximum ranges described above will be used by default unless overwritten using parameters, as is done in the code listing. The custom parameters defined in the metadata therefore overwrite the defaults such that each individual IQ array will have between 1 and 5 signals with SNRs between 0 dB and 40 dB. The range of signal center frequencies is -4 MHz and 4 MHz, the signal bandwidths will be between 100 kHz and 1 MHz and the burst durations will be have a minimum and maximum of 10 ms and 100 ms.

Metadata calls will need to be updated when upgrading from version 1.0 or 1.1 to 2.0. The first step is replacing *NarrowbandMetadata()* and *WidebandMetadata()* with *DatasetMetadata()*. These classes have many of the same parameters since *NarrowbandMetadata()* and *WidebandMetadata()* acted as wrappers for *DatasetMetadata()*.

3.3. Iterable Datasets

Previously, datasets were created with *NewNarrowband()* and *NewWideband()* which implemented the *__getitem__()* function. This allowed datasets to be created by indexing, for example *dataset = NewNarrowband(dataset_metadata)* and then calling *dataset[0]*, *dataset[1]*, *dataset[2]* and so on in sequential order. This has been replaced by an it-

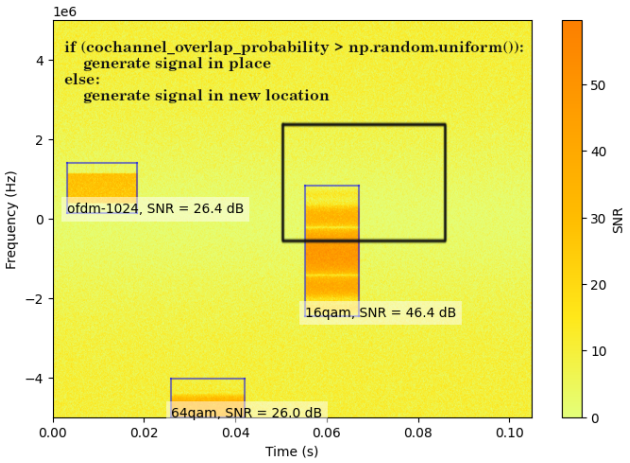


Figure 5. The co-channel overlap probability gives users the ability to minimize the amount of overlap in a dataset. It does not guarantee co-channel interference, it avoids it.

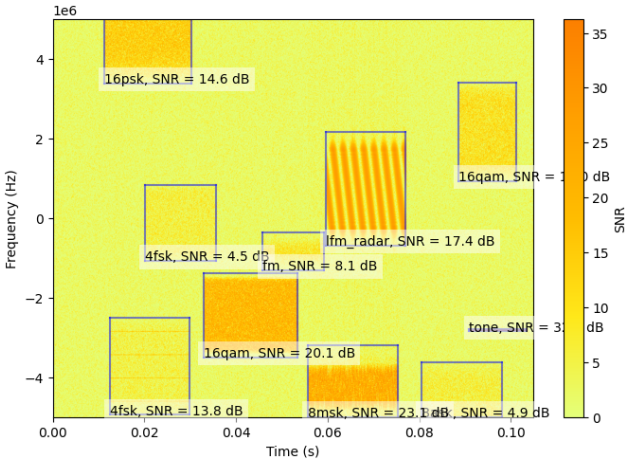


Figure 6. A spectrogram with no overlapping signals, *cochannel_overlap_probability = 0*.

erable class, *TorchSigIterableDataset*, since PyTorch's IterableDataset (PyTorch) better describes and supports the functions of a "new" dataset.

The dataset metadata from Listing 1 is passed into *TorchSigIterableDataset()* and each sample's data and metadata is created and returned using the *next()* operator as in Listing 2.

```
dataset = TorchSigIterableDataset(
    dataset_metadata = dataset_metadata)

data, metadata =
    next(TorchSigIterableDataset)
```

Listing 2. Creating a dataset and returning a sample.

3.4. Co-channel Interference Parameterization

The amount of overlap between signals, referred to as *co-channel interference*, is now controllable by setting the *cochannel_overlap_probability* parameter within *DatasetMetadata*. The parameter does not ensure co-channel interference but acts to prevent it. When a signal is to be generated whose bounding boxes are overlapping, the parameter determines the likelihood that the co-channel interferer remains. Figure 5 describes the logic for when a interfering signal can be placed.

Choosing *cochannel_overlap_probability = 0* ensures that no signal will have any co-channel interference (Figure 6), while *cochannel_overlap_probability = 1.0* allows for, but does not guarantee, co-channel interference.

3.5. Bring Your Own Data (BYOD)

Before version 2.0, the TorchSig pipeline exclusively supported TorchSig generated data. A highly requested feature–enabling users to apply TorchSig's framework to their own data–is now available in 2.0. However, users must write a bit of code to ensure compatibility with TorchSig.

The *ExternalFileHandler* class lets user data interface with TorchSig. It defines three methods that users must implement:

- *size* — returns the dataset size
- *load_dataset_metadata* — loads dataset metadata
- *load* — loads data and metadata

Users then pass their file handler class into *ExternalTorchSigDataset*, which works similarly to *StaticTorchSigDataset*. Transforms can be applied as usual.

```
class BYODExampleFileHandler(
    ExternalFileHandler
):
    ...

custom_dataset = ExternalTorchSigDataset(
    file_handler=
        BYODExampleFileHandler(root),
    transforms=[...],
)
```

Listing 3. BYOD Example

Note that users must supply all metadata required by the desired transforms. For instance, the *ChannelSwap* transform swaps I/Q channels to create frequency mirroring, and

updates the *center_freq* of the signal metadata. Thus, that metadata field must be present in order for the transform to apply correctly.

The BYOD feature enables users to leverage TorchSig in their RFML pipelines, where more robust machine learning models can train on augmented synthetic and real data, such as custom synthetic modulators, lab generated data or real-world IQ samples.

4. Future Work

The TorchSig 2.0 release focuses on improving the quality of synthetic datasets by adding in transforms that simulate non-linearities in RF hardware, giving users more flexibility and customization when creating datasets, and enabling users to bring their own data into TorchSig. For the future, we are focused on continuing our enhancement of user data pipelines and compatibility, and improving our synthetic signal generation with more realistic signals. We also aim to continue our active collaboration with the open-source community to innovate in the RFML space.

References

Boegner, L., Gulati, M., Vanhoy, G., Vallance, P., Comar, B., Kokalj-Filipovic, S., Lennon, C., and Miller, R. Large scale radio frequency signal classification, 2022a. URL https://arxiv.org/abs/2207.09918.

Boegner, L., Vanhoy, G., Vallance, P., Gulati, M., Feitzinger, D., Comar, B., and Miller, R. Large scale radio frequency wideband signal detection & recognition, 2022b. URL https://arxiv.org/abs/2211.10335.

fredric j. harris. *Multirate Signal Processing for Communication Systems: Second Edition*. River Publishers, Gistrup, Denmark, 2021.

Jupyter. Jupyter. https://jupyter.org.

Lyons, R. *Understanding Digital Signal Processing, Third Edition*. Prentice Hall, Upper Saddle River, NJ, 2011.

Oppenheim, A. and Schaefer, R. *Discrete-Time Signal Processing, Second Edition*. Prentice Hall, Upper Saddle River, NJ, 1999.

PyTorch. PyTorch: Iterable-style datasets. https://docs.pytorch.org/docs/stable/data.html#iterable-style-datasets.

TorchSig GitHub. TorchSig GitHub. https://github.com/torchdsp/torchsig.

TorchSig Website. Torchsig.com. https://torchsig.com/.

Ultralytics. Ultralytics. https://ultralytics.com.

Vallance, P., Oh, E., Mullins, J., Gulati, M., Hoffman, J., and Carrick, M. Torchsig: A gnu radio block and new spectrogram tools for augmenting ml training, 2024. URL https://events.gnuradio.org/event/24/contributions/628/.

Zarr. Zarr: chunked, compressed, N-dimensional arrays. https://zarr.dev.