digcommpy Documentation

Karl-Ludwig Besser

Jul 03, 2019

Contents:

This package provides functions for digital communication simulations.

CHAPTER 1

digcommpy

1.1 digcommpy package

1.1.1 Submodules

1.1.2 digcommpy.channels module

class digcommpy.channels.**AwgnChannel**(*snr_db*, *rate=1.0*, *input_power=None*)
Bases: *digcommpy.channels.Channel*

Additive white Gaussian noise channel.

Parameters

- **snr_db** (*float*) – Signal-to-noise ratio in dB.
- **rate** (*float, optional*) – Rate of the information bits to transmitted symbols (includes code and modulation rate)
- **input_power** (*float, optional*) – Input power of the symbols. If None, the power is estimated.

capacity()

get_channelstate()

set_params(*snr_db*)

transmit_data(*messages*)

class digcommpy.channels.**BawgnChannel**(**args*, ***kwargs*)
Bases: *digcommpy.channels.AwgnChannel*

capacity()

class digcommpy.channels.**BecChannel**(*epsilon*)
Bases: *digcommpy.channels.BinaryInputChannel*

Binary erasure channel.

capacity()

get_channelstate()

set_params(*epsilon*)

transmit_data(*messages*)

class digcommpy.channels.**BinaryInputChannel**(**args*, ***kwargs*)
Bases: *digcommpy.channels.Channel*, abc.ABC

Abstract channel class for binary input channels

transmit_data(*messages*)

class digcommpy.channels.**BscChannel**(*prob*)
Bases: *digcommpy.channels.BinaryInputChannel*

Binary symmetric channel

capacity()

get_channelstate()

set_params(*prob*)

transmit_data(*messages*)

class digcommpy.channels.**Channel**(**args*, ***kwargs*)
Bases: abc.ABC

Abstract channel class.

capacity()

get_channelstate()

set_params(*new_params*)

transmit_data(*messages*)

1.1.3 digcommpy.checks module

digcommpy.checks.**is_binary_message**(*message*)

1.1.4 digcommpy.decoders module

class digcommpy.decoders.**Decoder**(*code_length*, *info_length*, *base=2*, *parallel=True*)
Bases: abc.ABC

Abstract decoder class.

decode_messages(*messages*, *channel=None*)

class digcommpy.decoders.**ElmDecoder**(*code_length*, *info_length*, *neurons*, ***kwargs*)
Bases: *digcommpy.decoders.MachineLearningDecoder*

Decoder class which uses Extreme Learning Machines (ELM) for decoding.

The hpelm implementation is used.

Parameters **neurons** (*list of tuples*) – List of tuples, where each tuple looks like the following: (num_neuron, activation). The activation function can be either a string accepted by the ELM class or a numpy function. To be precise: the tuple is passed to the ELM.add_neurons method. Note that the output layer is linear.

decode_messages(*messages*, *channel=None*)

class digcommpy.decoders.**IdentityDecoder**(*code_length*, *info_length*, *base=2*, *parallel=True*)

Bases: *digcommpy.decoders.Decoder*

Identity decoder. Simply returns the input.

static decode_messages(*messages*, *channel=None*)

class digcommpy.decoders.**LinearDecoder**(*code_length*, *info_length*, *base=2*, *parallel=True*)

Bases: *digcommpy.decoders.Decoder*

Linear block decoder.

Parameters **TODO** –

decode_messages(*messages*, *channel=None*)

class digcommpy.decoders.**MachineLearningDecoder**(*code_length*, *info_length*, ***kwargs*)

Bases: *digcommpy.decoders.Decoder*

Decoder class using Machine Learning algorithms as decoder.

Parameters

- **code_length** (*int*) – Length of the block code.
- **info_length** (*int*) – Number of information bits.
- **training_data** (*list or tuple (arrays or [Encoder, Modulator])*) – Two arrays containing the codewords and corresponding messages (binary) or an Encoder class where the whole codebook is used as training data.

train_system(*training_data*, ***kwargs*)

Train the ML system using training data.

Parameters

- **training_data** (*list*) – List of objects to train the systems. Possible options are: *numpy arrays* in order: X, y or *Encoder*, *Modulator*. If the *Encoder* is used, all possible information words are generated and encoded.
- **kwargs** (*keyword arguments*) – All arguments that are accepted by the training method of the used algorithm.

class digcommpy.decoders.**NeuralNetDecoder**(*code_length*, *info_length*, *layer*, *train_snr=2.0*, *activation='relu'*, *one_hot=False*, *optimizer='adam'*, *loss='binary_crossentropy'*, ***kwargs*)

Bases: *digcommpy.decoders.MachineLearningDecoder*

Decoder class to decode channel codes using feed forward neural networks

Parameters

- **layer** (*list (int)*) – Number of nodes in each layer.
- **train_snr** (*float*) – Training SNR in dB.

- **activation** (*str*) – Activation function used for all hidden layers. See the Keras documentation for details.
- **loss** (*str, optional*) – Name of loss function. Default: 'binary_crossentropy'.
- **kwargs** (*keyword arguments*) – All arguements that are accepted by the Keras *keras.models.compile* method.

decode_messages(*messages*, *channel=None*)

class digcommpy.decoders.**PolarDecoder**(*code_length*, *info_length*, *design_channel*, *design_channelstate=0.0*, *pos_lookup=None*, *frozenbits=None*, *parallel=True*, ***kwargs*)

Bases: *digcommpy.decoders.Decoder*

Polar code decoder. Taken from **polarcodes.com**

The decoder for BAWGN channels expects a channel output of noisy codewords which are modulated to +1 and -1.

Parameters

- **code_length** (*int*) – Length of the code.
- **info_length** (*int*) – Length of the messages.
- **design_channel** (*str or Channel*) – Name of the used channel. Valid choices are currently "BAWGN" and "BSC".
- **design_channelstate** (*float, optional*) – State of the design channel. For "BAWGN" channels, this corresponds to the SNR value in dB. For "BSC" channels, this corresponds to the bit-flip probability.
- **pos_lookup** (*array, optional*) – Position lookup of the polar code, where -1 indicates message bits, while 0 and 1 denote the frozenbits.
- **frozenbits** (*array, optional*) – Bits used for the frozen bit positions. This is ignored, if *pos_lookup* is provided.
- **parallel** (*bool, optional*) – If True, parallel processing is used. This might not be available on all machines and causes higher use of system resources.

decode_messages(*messages*, *channel=None*)

Decode polar encoded messages.

Parameters

- **messages** (*array*) – Array of received (noisy) codewords which were created by polar encoding messages. Each row represents one received word.
- **channel** (*float or Channel, optional*) – This can either be a channel state, e.g., SNR in an AWGN channel, of the channel model used for constructing the decoder or a *channels.Channel* object. If None, the design parameters are used.

Returns decoded_messages – Array containing the estimated messages after decoding the channel output.

Return type array

class digcommpy.decoders.**PolarWiretapDecoder**(*code_length*, *design_channel_bob*, *design_channel_eve=None*, *design_channelstate_bob=0*, *design_channelstate_eve=0.0*, *pos_lookup=None*, *frozenbits=None*, *parallel=True*, *info_length_bob=None*, *random_length=None*, ***kwargs*)

Bases: *digcommpy.decoders.PolarDecoder*

Decoder class for decoding polar wiretap codes. You can either provide both channels (to Bob and Eve) or provide the main channel to Bob and the position lookup of the already constructed code.

Parameters

- **code_length** (*int*) – Length of the codewords.
- **design_channel_bob** (*str*) – Channel name of the main channel to Bob. Valid choices are the channel models which are supported by the PolarDecoder.
- **design_channel_eve** (*str, optional*) – Channel name of the side channel to Eve. Valid choices are the channel models which are supported by the PolarEncoder.
- **design_channelstate_bob** (*float, optional*) – Channelstate of the main channel.
- **design_channelstate_eve** (*float, optional*) – Channelstate of the side channel.
- **pos_lookup** (*array, optional*) – Position lookup of the constructed wiretap code. If this is provided, no additional code is constructed and the values of Eve's channel are ignored.

class digcommpy.decoders.**RepetitionDecoder**(**args*, ***kwargs*)

Bases: *digcommpy.decoders.Decoder*

static decode_messages(*messages*, *channel=None*)

class digcommpy.decoders.**SvmDecoder**(*code_length*, *info_length*, ***kwargs*)

Bases: *digcommpy.decoders.MachineLearningDecoder*

Decoder class which uses Support Vector Machines (SVM) for decoding.

The sklearn.svm implementation is used. All parameters which are accepted for creating a SVM can be used here.

Parameters

- **C** (*float, optional*) – Penalty score.
- **kernel** (*str, optional*) – Kernel function. See the sklearn.svm documentation for implemented kernel functions.
- **gamma** (*float, optional*) – Kernel parameter. This might be ignored, depending on the kernel.

decode_messages(*messages*, *channel=None*)

1.1.5 digcommpy.demodulators module

class digcommpy.demodulators.**BpskDemodulator**(**args*, ***kwargs*)

Bases: *digcommpy.demodulators.Demodulator*

static demodulate_symbols(*messages*)

class digcommpy.demodulators.**Demodulator**(**args*, ***kwargs*)
Bases: abc.ABC

Abstract modulator class

demodulate_symbols(*messages*)

class digcommpy.demodulators.**IdentityDemodulator**(**args*, ***kwargs*)
Bases: *digcommpy.demodulators.Demodulator*

static demodulate_symbols(*messages*)

class digcommpy.demodulators.**QamDemodulator**(**args*, ***kwargs*)
Bases: *digcommpy.demodulators.Demodulator*

static demodulate_symbols(*messages*, *m=4*)

1.1.6 digcommpy.encoders module

class digcommpy.encoders.**CodebookEncoder**(*code_length*, *info_length*, *codebook*, *wiretap=False*, *random_length=0*, ***kwargs*)
Bases: *digcommpy.encoders.Encoder*

Generic Encoder class for arbitrary codebooks.

This encoder allows encoding messages using a codebook as lookup table. Any mapping between messages and codewords may be used.

Parameters

- **codebook** (*tuple or dict or str*) – The mapping between messages and codewords. Either as a tuple of arrays (messages, codewords) where the rows correspond to each other, or as dict where the keys are the messages as integers, or as a path to a file containing a codebook.
- **wiretap** (*bool, optional*) – Set True if the code is a wiretap code.
- **random_length** (*int, optional*) – Number of random bits, if the code is a wiretap code. This is ignored, if wiretap is False.

encode_messages(*messages*)

generate_codebook()

class digcommpy.encoders.**Encoder**(*code_length*, *info_length*, *random_length=0*, *base=2*, *parallel=True*, ***kwargs*)
Bases: abc.ABC

Abstract encoder class

encode_messages(*messages*)

generate_codebook()

class digcommpy.encoders.**IdentityEncoder**(*code_length*, *info_length*, *random_length=0*, *base=2*, *parallel=True*, ***kwargs*)
Bases: *digcommpy.encoders.Encoder*

static encode_messages(*messages*)

class digcommpy.encoders.**LinearEncoder**(*code_matrix*, *base=2*, ***kwargs*)
Bases: *digcommpy.encoders.Encoder*

Linear block encoder.

Parameters

- **code_matrix** (*array*) – Code generation matrix. Dimension is (info_bits, code_length).
- **base** (*int, optional*) – Base of the field. Default is binary (2).

encode_messages(*messages*)

class digcommpy.encoders.**PolarEncoder**(*code_length*, *info_length*, *design_channel*, *design_channelstate=0.0*, *frozenbits=None*, *parallel=True*, ***kwargs*)

Bases: *digcommpy.encoders.Encoder*

Polar code encoder.

The implementation is copied from the Matlab implementation from http://www.polarcodes.com

Parameters

- **code_length** (*int*) – Length of the codewords.
- **info_length** (*int*) – Number of information bits.
- **design_channel** (*str or Channel object*) – Design channel name or channel object. Supported are: *AWGN*, *BEC* and *BSC*.
- **design_channelstate** (*float (optional)*) – State of the design channel. AWGN: SNR, BEC: epsilon, BSC: p.
- **frozenbits** (*array (optional)*) – Array of frozen bits. If not given, all zeros will be used.
- **parallel** (*bool (optional)*) – Use parallel encoding of the codewords. This might not be supported on all machines.

static construct_polar_code(*code_length*, *info_length*, *design_channel*, *design_channelstate*, *frozenbits=None*)

encode_messages(*messages*)

class digcommpy.encoders.**PolarWiretapEncoder**(*code_length*, *design_channel_bob*, *design_channel_eve*, *design_channelstate_bob=0*, *design_channelstate_eve=0*, *frozenbits=None*, *info_length_bob=None*, *random_length=None*, *parallel=True*, ***kwargs*)

Bases: *digcommpy.encoders.PolarEncoder*

Encoder for polar wiretap codes.

It uses [1] for the construction of the codes.

[1] H. Mahdavifar and A. Vardy, "Achieving the Secrecy Capacity of Wiretap Channels Using Polar Codes" IEEE Trans. Inf. Theory, vol. 57, no. 10, pp. 6428–6443, Oct. 2011.

Parameters

- **code_length** (*int*) – Length of the code
- **design_channel_bob** (*str or Channel object*) – Name of the design channel to Bob (main channel)
- **design_channel_eve** (*str or Channel object*) – Name of the design channel to Eve (wiretap channel)

- **design_channelstate_bob** (*float, optional*) – Design channelstate of the main channel. It is ignored if the *design_channel_bob* argument is a Channel like object.
- **design_channelstate_eve** (*float, optional*) – Design chanelstate of the wiretap channel. It is ignored if the *design_channel_eve* argument is a Channel like object.
- **frozenbits** (*array, optional*) – Array of frozen bits. If not given, all zeros will be used.
- **info_length_bob** (*int, optional*) – Number of information bits to Bob. If None, the number is calculated based on the main channel's capacity.
- **random_length** (*int, optional*) – Number of random bits. If None, the number is calculated based on the capacity of the eavesdropper's channel.

static construct_polar_wiretap_code(*code_length*, *design_channel_bob*, *design_channel_eve*, *design_channelstate_bob*, *design_channelstate_eve*, *frozenbits=None*, *info_length_bob=None*, *random_length=None*)

generate_codebook(*return_random=False*)

class digcommpy.encoders.**RepetitionEncoder**(*code_length*, ***kwargs*)

Bases: *digcommpy.encoders.Encoder*

encode_messages(*messages*)

1.1.7 digcommpy.information_theory module

class digcommpy.information_theory.**GaussianMixtureRv**(*mu*, *sigma=1.0*, *weights=None*)

Bases: object

Gaussian mixture random variable.

This class allows building Gaussian mixture random variables, where all components have the same covariance matrix.

Parameters

- **mu** (*array*) – List of the means of the individual Gaussian components. The shape therefore is (num_components, dimension).
- **sigma** (*array or float, optional*) – Covariance matrix. If only a float is provided, it is used for a scaled identity matrix as covariance matrix.
- **weights** (*list, optional*) – Weights/probabilities of the individual components. If None, a uniform distribution is used.

dim()

Return the dimension of the distribution

Returns dimension – Dimension of the distribution.

Return type int

logpdf(*x*)

pdf(*x*)

rvs(*N=1*)

digcommpy.information_theory.**binary_entropy**(*prob*)
Calculate the Shannon entropy of a binary random variable.

Parameters **prob** (*float*) – Probability of one event.

Returns **entr** – Entropy in bits.

Return type float

digcommpy.information_theory.**channel_capacity**(*channel*, *channelstate=None*)

digcommpy.information_theory.**entropy**(*prob*)
Calculate the Shannon entropy of a discrete random variable.

Parameters **prob** (*list (float)*) – List of probabilities of the random variable.

Returns **entr** – Entropy in bits.

Return type float

digcommpy.information_theory.**entropy_gauss_mix_lower**(*mu*, *sig*, *weights=None*, *alpha=0.5*)
Calculate a lower bound of the differential entropy of a Gaussian mixture.

Calculate a lower bound of the differential entropy of a Gaussian mixture using the Chernoff alpha-divergence as distance (alpha=.5 for Bhattacharyya distance) according to (Kolchinsky et al, 2017) (arXiv: 1706.02419).

Parameters

- **mu** (*array*) – Array containing the different means of the Gaussian mixture components. The shape is (num_components, dimensions).
- **sig** (*array*) – Covariance matrix of the components. It is the same for all components. If a float is provided, it is assumed as the noise variance for a scaled identity matrix as the covariance matrix.
- **weights** (*list, optional*) – Weights/probabilities of the individual mixture components. If None, a uniform distribution is used.
- **alpha** (*float, optional*) – Value used for the alpha-divergence. Default is 0.5 which uses the Bhattacharyya distance.

Returns **lower_bound_entropy** – Lower bound on the differential entropy of the Gaussian mixture.

Return type float

digcommpy.information_theory.**entropy_gauss_mix_upper**(*mu*, *sig*, *weights=None*)
Calculate an upper bound of the differential entropy of a Gaussian mixture.

Calculate an upper bound of the differential entropy of Gaussian mixture using the KL-divergence as distance according to (Kolchinsky et al, 2017) (arXiv: 1706.02419).

Parameters

- **mu** (*array*) – Array containing the different means of the Gaussian mixture components. The shape is (num_components, dimensions).
- **sig** (*array or float*) – Covariance matrix of the components. It is the same for all components. If a float is provided, it is assumed as the noise variance for a scaled identity matrix as the covariance matrix.
- **weights** (*list, optional*) – Weights/probabilities of the individual mixture components. If None, a uniform distribution is used.

Returns **upper_bound_entropy** – Upper bound on the differential entropy of the Gaussian mixture.

Return type float

digcommpy.information_theory.**get_info_length**(*code_length*, *channel*, *channelstate*)

1.1.8 digcommpy.messages module

digcommpy.messages.**generate_data**(*info_length*, *number=None*, *binary=False*)
Generate random messages.

Parameters

- **info_length** (*int*) – Number of information bits (message length)
- **number** (*int, optional*) – Number of random messages to generate (if int given) or all possible messages (if None given)
- **binary** (*bool, optional*) – If True, the messages will be returned in binary representation

Returns **messages** – Array of generated messages. Shape is (number, info_bits) if unpacked, (numer, 1) else.

Return type array

digcommpy.messages.**pack_to_dec**(*messages*)
This function converts an array of binary numbers into their decimal representation.

Parameters **messages** (*array (N x num_bits)*) – Array where each row contains one number and each column one bit

Returns **dec_messages** – Converted messages as decimal numbers

Return type array (N x 1)

digcommpy.messages.**unpack_to_bits**(*messages*, *num_bits*)
This function converts an array of dec numbers into their binary representation.

Parameters

- **messages** (*list*) – List of messages (numbers)
- **num_bits** (*int*) – Number of output bits

Returns **binary_messages** – Converted messages as bits

Return type array (N x num_bits)

1.1.9 digcommpy.metrics module

digcommpy.metrics.**ber**(*y_true*, *y_pred*, *normalize=True*)
Calculate the Bit Error Ratio (BER) between two arrays.

Parameters

- **y_true** (*numpy.array*) – Array with the true bits. The shape is [num_messages, num_bits].
- **y_pred** (*numpy.array*) – Array with the predicted bits. Same shape as *y_true*.
- **normalize** (*bool, optional*) – If True, the results is normalized to be between 0 and 1. Otherwise, the number of errors is returned.

Returns **ber** – Bit error ratio or number of bit errors

Return type float

digcommpy.metrics.**bler**(*y_true*, *y_pred*, *normalize=True*)
Calculate the Block Error Ratio (BLER) between two arrays.

Parameters

- **y_true** (*numpy.array*) – Array with the true bits. The shape is [num_messages, num_bits].
- **y_pred** (*numpy.array*) – Array with the predicted bits. Same shape as *y_true*.
- **normalize** (*bool, optional*) – If True, the results is normalized to be between 0 and 1. Otherwise, the number of errors is returned.

Returns **bler** – Block error ratio

Return type float

1.1.10 digcommpy.modulators module

class digcommpy.modulators.**BpskModulator**(**args*, ***kwargs*)
Bases: *digcommpy.modulators.Modulator*

static modulate_symbols(*messages*)

class digcommpy.modulators.**IdentityModulator**(**args*, ***kwargs*)
Bases: *digcommpy.modulators.Modulator*

static demodulate_symbols(*messages*)

static modulate_symbols(*messages*)

class digcommpy.modulators.**Modulator**(**args*, ***kwargs*)
Bases: abc.ABC

Abstract modulator class

modulate_symbols(*messages*)

class digcommpy.modulators.**QamModulator**(**args*, ***kwargs*)
Bases: *digcommpy.modulators.Modulator*

static modulate_symbols(*messages*, *m=4*)
Modulate binary messages or codewords with QAM.

Parameters

- **messages** (*array*) – Array of messages or codewords that should be modulated. Every row corresponds to one individual message. The number of columns is the length of the codewords and has to be an integer multiple of *m*.
- **m** (*int, optional*) – Modulation order. It has to be a square of a power of two.

Returns **symbols** – Array of modulated symbols. The number of rows is the same as in *messages*. The number of rows is divided by *m*.

Return type array

1.1.11 digcommpy.parsers module

digcommpy.parsers.**convert_codebook_to_dict**(*messages*, *codewords*, *random=None*)
Convert a codebook representation to dictionary.

Convert the codebook representation of multiple arrays to a single dict.

Parameters

- **messages** (*array*) – Array of the messages, where each row represents one message.
- **codewords** (*array*) – Array of the codewords corresponding to the messages.
- **random** (*array, optional*) – Optional array of random bits which is used for wiretap codes.

Returns

- **codebook** (*dict*) – Codebook dictionary where the keys are the messages as decimal numbers.
- **code_info** (*dict*) – Dict of different code parameters.

digcommpy.parsers.**read_codebook_file**(*filename*, *wiretap=False*, *columns=None*, ***kwargs*)

Read a codebook file.

Read a file which contains the codebook in columns. The default expected column names are *message* and *codeword*.

Parameters

- **filename** (*str*) – File path to the file.
- **wiretap** (*bool, optional*) – Set to True if the codebook is from a wiretap code.
- **columns** (*dict, optional*) – If provided, the entries are used as column names. The supported keys are *message*, *codeword*, and *random* for wiretap codes.
- ****kwargs** (*keyword-arguments, optional*) – All kwargs that can be passed to the pd.read_csv function.

Returns

- **codebook** (*dict*) – Mapping of the messages to the codewords.
- **code_info** (*dict*) – Dict of different code parameters.

digcommpy.parsers.**read_hyperparameter_search_results**(*filename*)

Read a results file from a HyperparameterSearchDecoderSimulation.

Parameters **filename** (*str*) – File path or file object

Returns

- **constants** (*dict*) – Dict containing all the constants of the simulation.
- **results** (*list*) – List of all simulation results for the evaluated hyperparameter configurations.

digcommpy.parsers.**read_simulation_log**(*filename*)

Read a results file from a CustomSimulation.

Parameters **filename** (*str*) – File path or file object

Returns **results** – Dict of all the results as returned from the original simulation.

Return type dict

1.1.12 digcommpy.simulations module

class digcommpy.simulations.**ChannelParameterSimulation**(*encoder*, *decoder*, *channel*, *modulator=<class 'digcommpy.modulators.IdentityModulator'>*, *demodulator=<class 'digcommpy.demodulators.IdentityDemodulator'>*, *logger=None*)

Bases: object

Generic class for simulations with different channel parameters.

Use this class for common simulations like SNR-BER simulations. The code parameters as well as the encoder and decoder are held constant and only the channel parameters are kept constant.

Parameters

- **encoder** (Encoder) – Encoder object used for encoding messages.
- **decoder** (Decoder) – Decoder object used for decoding the demodulator output.
- **channel** (Channel) – Channel object used to corrupt the transmitted symbols with noise
- **modulator** (Modulator, *optional*) – Modulator object used to modulate the codewords before transmitting. Default is to use no modulation.
- **demodulator** (Demodulator, *optional*) – Demodulator object used to demodulate the channel output before trying to decode it. Default is to use no demodulation.
- **logger** (*logging.Logger*, *optional*) – Logger object which is used to log information about the simulation.

simulate(*test_params*, *test_size=1000000.0*, *metric=['bler', 'ber']*, *batch_size=50000*)

Run a simulation with provided options.

Parameters

- **test_params** (*list*) – List of simulation variables.
- **test_size** (*int*, *optional*) – Number of test messages.
- **metric** (*list (str)*, *optional*) – List of metrics that are calculated. Possible choices are "ber" for the bit error rate and "bler" for the block error rate.
- **batch_size** (*int*, *optional*) – Size of the test batches.

Returns **results** – Dict including all the results (metrics) for the evaluated simulation variables.

Return type dict

class digcommpy.simulations.**CustomSimulation**(*encoder*, *decoder*, *channel*, *modulator=<class 'digcommpy.modulators.IdentityModulator'>*, *demodulator=<class 'digcommpy.demodulators.IdentityDemodulator'>*, *logger=None*)

Bases: object

Fully customizable tranmission simulation.

Parameters

- **encoder** (Encoder) – Class object of Encoder like class
- **decoder** (Decoder) – Class object of Decoder like class

- **channel** (Channel) – Class object of Channel like class
- **modulator** (Modulator, *optional*) – Class object of Modulator like class
- **demodulator** (Demodulator, *optional*) – Class object of Demodulator like class
- **logger** (*Logger, optional*) – Logger object from logging package

simulate(*simulation_parameters*, *channel_options*, *enc_opt=None*, *dec_opt=None*, *mod_opt=None*, *demod_opt=None*, *training_opt=None*, ***kwargs*)

class digcommpy.simulations.**HyperparameterSearchDecoderSimulation**(*encoder*, *decoder_class*, *channel_class*, *decoder_variables*, *channel_variables*, *modulator=<class 'digcommpy.modulators.IdentityModula*, *demodulator=<class 'digcommpy.demodulators.IdentityDemo*, *logger=None*)

Bases: object

Class for a hyperparameter grid search for a machine learning decoder.

The system is assumed to have a constant encoder, modulator and demodulator. The hyperparameters of the decoder system will be adjusted and tested for different channel parameters.

Parameters

- **encoder** (encoders.Encoder) – Encoder object which will be used to generate the codewords.
- **decoder_class** (*decoders.MachineLearningDecoder <class>*) – Decoder class which will be instantiated with the *decoder_variables*.
- **channel_class** (*channels.Channel <class>*) – Channel class which will be instantiated with the *channel_variables*.
- **decoder_variables** (*dict*) – Dict containing the hyperparameters of the decoder to be varied. The dict will be used to create a grid of all possible combinations.
- **channel_variables** (*dict*) – Dict containing the parameters of the channel to be varied for each evaluation. The dict will be used to create a grid of all possible combinations.
- **modulator** (modulators.Modulator, *optional*) – Modulator object which will be used to modulate the codewords.
- **demodulator** (demodulators.Demodulator, *optional*) – Demodulator object which will be used to demodulate the codewords.
- **logger** (*logging.Logger, optional*) – Logger object which is used for the simulation output. If None, a default one will be used.

static create_logger(*filename=None*)

start_simulation(*test_size=1000, metric=['ber', 'bler'], training_options=None, seed=None*)
Start the full simulation with all possible settings.

Parameters

- **test_size** (*int, optional*) – Number of test messages for evaluation.
- **metric** (*list, optional*) – List of metrics that should be calculated. Valid choices are *ber* and *bler*.
- **training_options** (*dict, optional*) – Keyword arguments, which are passed to MachineLearningDecoder.train_system.
- **seed** (*int, optional*) – Seed for initializing the test set. This is only used for the data generation of the test messages. If None, a random seed will be used.

Returns **simulation_results** – List of simulation results. Each element is a tuple of the evaluated hyperparameters and the simulation results.

Return type list

digcommpy.simulations.**create_simulation_var_combinations**(*sim_var_params*)
Generate a simulation variable dictionary for a CustomSimulation

Parameters **sim_var_params** (*dict*) – Dict containing the different variable names as keys and their values as values in a list.

Returns **sim_var** – List of all different simulation parameters.

Return type list

digcommpy.simulations.**single_simulation**(*encoder, decoder, channel, modulator=<class 'digcommpy.modulators.IdentityModulator'>, demodulator=<class 'digcommpy.demodulators.IdentityDemodulator'>, metric=['ber', 'bler'], test_size=1000000.0, batch_size=50000, test_data_generator=None, seed=None*)
Run a single simulation with fixed parameters.

Parameters

- **encoder** (Encoder) – Encoder instance which is used for encoding messages.
- **decoder** (Decoder) – Decoder instance which is used for decoding messages.
- **channel** (Channel) – Channel instance which is used for corrupting the transmitted messages.
- **modulator** (Modulator, *optional*) – Modulator instance which is used for modulating the codewords before transmission. The default is no modulation.
- **demodulator** (Demodulator, *optional*) – Demodulator instance which is used for demodulating the channel output before decoding. The default is no demodulation.
- **metric** (*list, optional*) – List of metrics which are calculated and returned.
- **test_size** (*int, optional*) – Number of messages to be tested.
- **batch_size** (*int, optional*) – The number of messages which are processed within one batch. Increasing this number may cause memory issues.
- **test_data_generator** (*generator, optional*) – Provide a generator instance which return the test data. Overwrites the test_size and batch_size keyword.

- **seed** (*int, optional*) – Seed which is used for the test_data_generator. If None, a random one, will be used.

Returns **results** – Dict containing all simulation results. The keys are the metrics and the values are the corresponding metric value.

Return type dict

1.1.13 Module contents

CHAPTER 2

Indices and tables

- genindex
- modindex
- search

Python Module Index

d

Index

E

G

H

I

L

M