Using pyAerial for LDPC encoding-decoding chain

Table of contents

This example shows how to use the pyAerial Python bindings to run 5G NR LDPC encoding, rate matching and decoding. Information bits, i.e. a transport block, get segmented into code blocks, LDPC encoded and rate matched onto the available time-frequency resources (resource elements), all following TS 38.212 precisely. The bits are then transmitted over an AWGN channel using QPSK modulation. At the receiver side, log likelihood ratios are extracted from the received symbols, (de)rate matching is performed and LDPC decoder is run to get the transmitted information bits. Finally, the code blocks are concatenated back into a received transport block.

pyAerial utilizes the cuPHY library underneath for all components, except code block segmentation and concatenation are currently written in Python. Also, CRCs are just random blocks of bits in this example as we can compare the transmitted and received bits directly to compute block error rates.

The NVIDIA Sionna library is utilized for simulating the radio channel.

Imports

[1]:

```
%matplotlib widget from cuda import cudart from collections import defaultdict import datetime import os os.environ["CUDA_VISIBLE_DEVICES"] = "0" os.environ['TF_CPP_MIN_LOG_LEVEL'] = "3" # Silence TensorFlow. import numpy as np import sionna import tensorflow as tf import matplotlib.pyplot as plt from aerial.phy5g.ldpc import LdpcEncoder from aerial.phy5g.ldpc import LdpcDecoder from aerial.phy5g.ldpc import LdpcRateMatch from aerial.phy5g.ldpc import LdpcDeRateMatch from aerial.phy5g.ldpc import get_mcs from aerial.phy5g.ldpc import random_tb from aerial.phy5g.ldpc import code_block_segment from aerial.phy5g.ldpc import code_block_desegment from aerial.phy5g.ldpc import get_crc_len # Configure the notebook to use only a single GPU and allocate only as much memory as needed. # For more details, see https://www.tensorflow.org/guide/gpu. gpus = tf.config.list_physical_devices('GPU') tf.config.experimental.set_memory_growth(gpus[0], True) from tensorflow.python.ops.numpy_ops import np_config np_config.enable_numpy_behavior()
```

Parameters

Set simulation parameters, some numerology parameters, enable/disable scrambling etc.

[2]:

```
# Simulation parameters. esno_db_range = np.arange(2.8, 3.5, 0.1) num_slots =
10000 min_num_tb_errors = 250 # Numerology and frame structure. See TS 38.211.
num_prb = 100 # Number of allocated PRBs. This is used to compute the transport
block # as well as the rate matching length. start_sym = 0 # PxSCH start symbol
num_symbols = 14 # Number of symbols in a slot. num_slots_per_frame = 20 #
Number of slots in a single frame. num_layers = 1 dmrs_sym = [0, 0, 1, 0, 0, 0, 0, 0, 0,
0, 0, 1, 0, 0] # Rate matching procedure includes scrambling if this flag is set.
enable_scrambling = True # The scrambling initialization value is computed as per TS
38.211 # using the RNTI and data scrambling ID. rnti = 20000 # UE RNTI data_scid = 41
# Data scrambling ID cinit = (rnti << 15) + data_scid rv = 0 # Redundancy version mcs =
10 # MCS index as per TS 38.214 table. mod_order, code_rate = get_mcs(mcs)
code_rate /= 1024.
```

Helper class for simulation monitoring

This helper class plots the simulation results and shows simulation progress in a table.

[3]:

```
class SimulationMonitor: """Helper class to show the progress and results of the
simulation.""" markers = ["d", "o", "s"] linestyles = ["-", "--", ":"] colors = ["blue",
"black", "red"] def __init__(self, cases, esno_db_range): """Initialize the
SimulationMonitor. Initialize the figure and the results table. """ self.cases = cases
self.esno_db_range = esno_db_range self.current_esno_db_range = [] self.start_time
= None self.esno_db = None self.bler = defaultdict(list) self._print_headers() def
step(self, esno_db): """Start next Es/No value.""" self.start_time =
datetime.datetime.now() self.esno_db = esno_db
self.current_esno_db_range.append(esno_db) def update(self, num_tbs,
num_tb_errors): """Update current state for the current Es/No value."""
```

```
self._print_status(num_tbs, num_tb_errors, False) def _print_headers(self): """Print result table headers."""" cases_str = " " * 21 separator = " " * 21 for case in self.cases: cases_str += case.center(20) + " " separator += "-" * 20 + " " print(cases_str) print(separator) title_str = "Es/No (dB)".rjust(12) + "TBs".rjust(8) + " " for case in self.cases: title_str += "TB Errors".rjust(12) + "BLER".rjust(8) + " " title_str += "ms/TB".rjust(8) print(title_str) print(("=" * 20) + " " + ("=" * 20 + " ") * len(self.cases) + "=" * 8) def _print_status(self, num_tbs, num_tb_errors, finish): """Print simulation status in a table."""" end_time = datetime.datetime.now() t_delta = end_time - self.start_time if finish: newline_char = '\n' else: newline_char = '\r' result_str = f" {self.esno_db:9.2f}".rjust(12) + f"{num_tbs:8d}".rjust(8) + " " for case in self.cases: result_str += f"{num_tb_errors[case]:8d}".rjust(12) result_str += f" {(num_tb_errors[case] / num_tbs):.4f}".rjust(8) + " " result_str += f" {(t_delta.total_seconds() * 1000 / num_tbs):6.1f}".rjust(8) print(result_str, end=newline_char) def finish_step(self, num_tbs, num_tb_errors): """Finish simulating the current Es/No value and add the result in the plot."""" self._print_status(num_tbs, num_tb_errors, True) for case_idx, case in enumerate(self.cases): self.bler[case].append(num_tb_errors[case] / num_tbs) def finish(self): """Finish simulation and plot the results."""" self.fig = plt.figure() for case_idx, case in enumerate(self.cases): plt.plot( self.current_esno_db_range, self.bler[case], marker=SimulationMonitor.markers[case_idx], linestyle=SimulationMonitor.linestyles[case_idx], color=SimulationMonitor.colors[case_idx], markersize=8, label=case ) plt.yscale('log') plt.ylim(0.001, 1) plt.xlim(np.min(self.esno_db_range), np.max(self.esno_db_range)) plt.title("Receiver BLER Performance vs. Es/No") plt.ylabel("BLER") plt.xlabel("Es/No [dB]") plt.grid() plt.legend() plt.show()
```

Create the LDPC coding chain objects

The LDPC coding chain objects are created here. This includes the following: * `LdpcEncoder` which takes the information bits, i.e. the transport block, segmented into code blocks as its input, and outputs encoded code blocks. * `LdpcRateMatch` which takes encoded code blocks as its input and outputs a rate matched (and optionally scrambled) stream of bits. * `LdpcDerateMatch` which takes the received stream of log-likelihood ratios (LLRs) as its input and outputs derate matched code blocks of LLRs which can be fed to the LDPC decoding. This block performs also descrambling if scrambling is enabled in the pipeline. * `LdpcDecoder` which takes the output of LDPC derate matching

and decodes the LLRs into code blocks that can then be further concatenated into a received transport block.

All components are based on TS 38.212 and thus can be used for transmitting/receiving 5G NR compliant bit streams.

Also the Sionna channel components and modulation mapper are created here.

[4]:

 # Create also the CUDA stream that running the objects requires. cudart.cudaSetDevice(0) cuda_stream = cudart.cudaStreamCreate()[1] cudart.cudaStreamSynchronize(cuda_stream) *# Create the Aerial Python LDPC objects.* ldpc_encoder = LdpcEncoder(cuda_stream=cuda_stream) ldpc_decoder = LdpcDecoder(half_precision=True, cuda_stream=cuda_stream) ldpc_rate_match = LdpcRateMatch(enable_scrambling=enable_scrambling, cuda_stream=cuda_stream) ldpc_derate_match = LdpcDeRateMatch(enable_scrambling=enable_scrambling, cuda_stream=cuda_stream) *# Create the Sionna modulation mapper/demapper and the AWGN channel.* mapper = sionna.mapping.Mapper("qam", 2) demapper = sionna.mapping.Demapper("app", "qam", 2) channel = sionna.channel.AWGN()

[5]:

case = "LDPC decoding perf." monitor = SimulationMonitor([case], esno_db_range) *# Loop the Es/No range.* for esno_db in esno_db_range: monitor.step(esno_db) num_tb_errors = defaultdict(int) *# Run multiple slots and compute BLER.* for slot_idx in range(num_slots): slot_number = slot_idx % num_slots_per_frame *# Generate a random transport block (in bits).* transport_block = random_tb(mod_order=mod_order, code_rate=code_rate * 1024, dmrs_syms=dmrs_sym, num_prbs=num_prb, start_sym=start_sym, num_symbols=num_symbols, num_layers=num_layers, return_bits=True) tb_size = transport_block.shape[0] *# Attach a CRC. This is emulated to get the TB size with CRC right, however the CRC is in this case just random # bits as we are comparing the transmitted and received bits directly to get the BLER (instead of doing an actual # CRC check).* crc_length = get_crc_len(tb_size) crc = np.random.randint(0, 1, size=crc_length, dtype=np.uint8) transport_block = np.concatenate((transport_block, crc)) *# Code block segmentation happens here. Note: This is just Python at the moment.* code_blocks =

code_block_segment(tb_size, transport_block, code_rate) *# Run the LDPC encoding. The LDPC encoder takes a K x C array as its input, where K is the number of bits per code # block and C is the number of code blocks. Its output is N x C where N is the number of coded bits per code block. # If there is more than one code block, a code block CRC (random in this case as we do not need an actual CRC) is # attached to* coded_bits = ldpc_encoder.encode(input_data=code_blocks, tb_size=tb_size, code_rate=code_rate, redundancy_version=rv) *# Run rate matching. This needs rate matching length as its input, meaning the number of bits that can be # transmitted within the allocated resource elements. The input data is fed as 32-bit floats.* num_data_sym = (np.array(dmrs_sym[start_sym:start_sym + num_symbols]) == 0).sum() rate_match_len = num_data_sym * num_prb * 12 * num_layers * mod_order rate_matched_bits = ldpc_rate_match.rate_match(input_data=coded_bits, tb_size=tb_size, code_rate=code_rate, rate_match_len=rate_match_len, mod_order=mod_order, num_layers=num_layers, redundancy_version=rv, cinit=cinit) *# Map the bits to symbols and transmit through an AWGN channel. All this in Sionna.* rate_matched_bits = rate_matched_bits[:, 0] no = sionna.utils.ebnodb2no(esno_db, num_bits_per_symbol=1, coderate=1) tx_symbols = mapper(rate_matched_bits[None]) rx_symbols = channel([tx_symbols, no]) llr = -1. * demapper([rx_symbols, no])[0, :].numpy()[:, None] *# Run receiver side (de)rate matching. The input is the received array of bits directly, and the output # is a NumPy array of size N x C of log likelihood ratios, represented as 32-bit floats. Descrambling # is also performed here in case scrambling is enabled.* derate_matched_bits = ldpc_derate_match.derate_match(input_data=llr, tb_size=tb_size, code_rate=code_rate, rate_match_len=rate_match_len, mod_order=mod_order, num_layers=num_layers, redundancy_version=rv, ndi=1, cinit=cinit) *# Run LDPC decoding. The decoder takes the derate matching output as its input and returns* decoded_bits = ldpc_decoder.decode(input_llr=derate_matched_bits, tb_size=tb_size, code_rate=code_rate, redundancy_version=rv, rate_match_len=rate_match_len) decoded_tb = code_block_desegment(decoded_bits, tb_size, code_rate) tb_error = not np.array_equal(decoded_tb[:-24], transport_block[:-24]) num_tb_errors[case] += tb_error monitor.update(num_tbs=slot_idx + 1, num_tb_errors=num_tb_errors) if (np.array(list(num_tb_errors.values())) >= min_num_tb_errors).all(): break *# Next Es/No value.* monitor.finish_step(num_tbs=slot_idx + 1, num_tb_errors=num_tb_errors) monitor.finish()

```
LDPC decoding perf. ---------- Es/No (dB) TBs TB Errors BLER ms/TB
========== ========== ======== 2.80 250 250 1.0000 20.5 2.90 250 250 1.0000 17.6 3.00 250 250 1.0000 17.5 3.10 258 250 0.9690 17.1 3.20 409 250 0.6112 16.8 3.30 2071 250 0.1207 17.0 3.40 10000 101 0.0101 16.9 3.50 10000 5 0.0005 17.0
```