Using pyAerial for data generation by simulation

Table of contents

This notebook generates a fully 5G NR compliant PUSCH/PDSCH dataset using NVIDIA cuPHY through its Python bindings in pyAerial for PUSCH/PDSCH slot generation and NVIDIA Sionna for radio channel modeling. PUSCH/PDSCH slots get generated and transmitted through different radio channels. Usually, in order to make models as generalizable as possible, it is desirable to train the models with as wide variety of different channel models as possible. This notebook enables generation of a dataset containing samples generated with a number of different channel models, including e.g. those used by 3GPP, as well as with different MCS classes and other transmission parameters.

Imports

[1]:

import warnings warnings.filterwarnings('ignore') import itertools import os os.environ["CUDA_VISIBLE_DEVICES"] = "0" os.environ['TF_CPP_MIN_LOG_LEVEL'] = "3" *# Silence TensorFlow.* import numpy as np import pandas as pd import sionna import tensorflow as tf from tqdm.notebook import tqdm from aerial.phy5g.pdsch import PdschTx from aerial.phy5g.ldpc.util import get_mcs, random_tb from aerial.util.fapi import dmrs_bit_array_to_fapi from aerial.util.data import PuschRecord from aerial.util.data import save_pickle *# This is for Sionna and pyAerial to coexist on the same GPU: # 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)

Dataset generation parameters

The parameters used to generate the dataset are modified here. Note that some parameters are given as lists, meaning that multiple values may be given for those parameters. Typically one would like the training dataset to be as diverse as possible in order to make the models generalize well to various channel conditions and to different transmission parameters.

[2]:

```
# This is the target directory. It gets created if it does not exist. dataset_dir =
'data/example_simulated_dataset/QPSK' os.makedirs(dataset_dir, exist_ok=True) #
Number of samples is divided roughly evenly between the options below. num_samples
= 12000 # A list of channel models: Suitable values: # "Rayleigh" - Rayleigh block fading
channel model (sionna.channel.RayleighBlockFading) # "CDL-x", where x is one of ["A",
"B", "C", "D", "E"] - for 3GPP CDL channel models # as per TR 38.901. channel_models =
["CDL-D"] # Speeds to include in the dataset # This is UE speed in m/s. The direction of
travel will be chosen randomly within the x-y plane. speeds = [0.8333] # Delay spreads
to include in the dataset. # This is the nominal delay spread in [s]. Please see the CDL
documentation # about how to choose this value. delay_spreads = [100e-9] # A list of
MCS indices (as per TS 38.214) to include in the dataset. # MCS table value refers to TS
38.214 as follows: # 1: TS38.214, table 5.1.3.1-1. # 2: TS38.214, table 5.1.3.1-2. # 3:
TS38.214, table 5.1.3.1-3. mcss = [1] # 1, 10, 19 used for QPSK, 16QAM and 64QAM,
respectively. mcs_table = 2 # Es/No values to include in the dataset. # esnos = [9.0, 9.25,
9.5, 9.75, 10.0, 10.25, 10.5, 10.75, 11.0] # MCS 19 # esnos = [-0.5, -0.25, 0.0, 0.25, 0.5,
0.75, 1.0, 1.25, 1.5, 1.75, 2.0, 2.25, 2.5, 2.75, 3.0] # MCS 10 esnos = [-7.75, -7.5, -7.25,
-7.0, -6.75, -6.5] # MCS 1 # These are fixed for the dataset. num_tx_ant = 1 num_rx_ant
= 4 cell_id = 41 carrier_frequency = 3.5e9 # Carrier frequency in Hz. link_direction =
"uplink" layers = 1 rnti = 20001 scid = 0 data_scid = 41 dmrs_port = 1 dmrs_position
= [0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0] start_sym = 0 num_symbols = 14 start_prb = 0
num_prbs = 273 # Numerology and frame structure. See TS 38.211. mu = 1
num_ofdm_symbols = 14 fft_size = 4096 cyclic_prefix_length = 288
subcarrier_spacing = 30e3 num_guard_subcarriers = (410, 410)
num_slots_per_frame = 20
```

Channel generation

Radio channel generation is done using NVIDIA Sionna.

[3]:

```
class Channel(sionna.channel.OFDMChannel): def __init__(self, link_direction,
channel_model, num_tx_ant, num_rx_ant, carrier_frequency, delay_spread, speed,
resource_grid): self.resource_grid = resource_grid self.resource_grid_mapper =
sionna.ofdm.ResourceGridMapper(resource_grid) self.remove_guard_subcarriers =
sionna.ofdm.RemoveNulledSubcarriers(resource_grid) # Define the antenna arrays.
ue_array = sionna.channel.tr38901.Antenna( polarization="single",
polarization_type="V", antenna_pattern="38.901",
carrier_frequency=carrier_frequency ) gnb_array =
sionna.channel.tr38901.AntennaArray( num_rows=1, num_cols=int(num_rx_ant/2),
polarization="dual", polarization_type="cross", antenna_pattern="38.901",
carrier_frequency=carrier_frequency ) if channel_model == "Rayleigh": ch_model =
sionna.channel.RayleighBlockFading( num_rx=1, num_rx_ant=num_rx_ant,
num_tx=1, num_tx_ant=num_tx_ant ) elif "CDL" in channel_model: cdl_model =
channel_model[-1] # Configure a channel impulse reponse (CIR) generator for the CDL
model. ch_model = sionna.channel.tr38901.CDL( cdl_model, delay_spread,
carrier_frequency, ue_array, gnb_array, link_direction, min_speed=speed ) else: raise
ValueError(f"Invalid channel model{channel_model}!") super().__init__( ch_model,
resource_grid, add_awgn=True, normalize_channel=True, return_channel=False )
def __call__(self, tx_tensor, No): # Add batch and num_tx dimensions that Sionna
expects and reshape. tx_tensor = tf.transpose(tx_tensor, (2, 1, 0)) tx_tensor =
tf.reshape(tx_tensor, (1, -1))[None, None] tx_tensor =
self.resource_grid_mapper(tx_tensor) rx_tensor = super().__call__((tx_tensor, No))
rx_tensor = self.remove_guard_subcarriers(rx_tensor) rx_tensor = rx_tensor[0, 0]
rx_tensor = tf.transpose(rx_tensor, (2, 1, 0)) return rx_tensor # Define the resource
grid. resource_grid = sionna.ofdm.ResourceGrid(
num_ofdm_symbols=num_ofdm_symbols, fft_size=fft_size,
subcarrier_spacing=subcarrier_spacing, num_tx=1, num_streams_per_tx=1,
cyclic_prefix_length=cyclic_prefix_length,
num_guard_carriers=num_guard_subcarriers, dc_null=False, pilot_pattern=None,
pilot_ofdm_symbol_indices=None )
```

PDSCH transmitter

This creates the PDSCH transmitter. However due to the symmetry of 5G NR PDSCH and PUSCH, this may be used to generate also PUSCH frames with certain parameterization. In this notebook this is used as a PUSCH transmitter to generate uplink slots.

```
[4]:
```

```
pxsch_tx = PdschTx( cell_id=cell_id, num_rx_ant=num_tx_ant,
num_tx_ant=num_tx_ant, )
```

Dataset generation

The actual dataset generation is done here. The different channel, SNR and MCS parameters are swept through, with a number of samples per parameterization chosen such that the total number of samples will be close to the desired number.

The PxSCH transmitter created above is used to generate a Tx frame. This Tx frame is then fed through the Sionna-generated radio channel. The resulting data is recorded in a Parquet file containing PUSCH records following roughly the Small Cell Forum FAPI specification format.

```
[5]:
```

```
num_cases = len(channel_models) * len(esnos) * len(speeds) * len(delay_spreads) *
len(mcss) num_samples_per_param = num_samples // num_cases # loop different
channel models, speeds, delay spreads, MCS levels etc. pusch_records = [] for
(channel_model, esno, speed, delay_spread, mcs) in \ (pbar :=
tqdm(itertools.product(channel_models, esnos, speeds, delay_spreads, mcss),
total=num_cases)): status_str = f"Generating... ({channel_model}|{esno}dB |
{speed}m/s |{delay_spread}s | MCS{mcs})" pbar.set_description(status_str) # Create
the channel model. channel = Channel( link_direction=link_direction,
channel_model=channel_model, num_tx_ant=num_tx_ant, num_rx_ant=num_rx_ant,
carrier_frequency=carrier_frequency, delay_spread=delay_spread, speed=speed,
resource_grid=resource_grid ) for sample in range(num_samples_per_param): #
Generate the dataframe. slot_number = sample % num_slots_per_frame # Get
modulation order and coderate. mod_order, coderate = get_mcs(mcs, mcs_table)
tb_input = random_tb(mod_order, coderate, dmrs_position, num_prbs, start_sym,
num_symbols, layers) # Transmit PxSCH. This is where we set the dynamically changing
parameters. # Input parameters are given as lists as the interface supports multiple UEs.
tx_tensor = pxsch_tx.run( tb_inputs=[tb_input], # Input transport block in bytes.
num_ues=1, # We simulate only one UE here. slot=slot_number, # Slot number.
dmrs_syms=dmrs_position, # List of binary numbers indicating which symbols are
DMRS. start_sym=start_sym, # Start symbol index. num_symbols=num_symbols, #
Number of symbols. scids=[scid], # DMRS scrambling ID. layers=[layers], # Number of
layers (transmission rank). dmrs_ports=[dmrs_port], # DMRS port(s) to be used. rntis=
[rnti], # UE RNTI. data_scids=[data_scid], # Data scrambling ID. code_rates=
[coderate], # Code rate mod_orders=[mod_order] # Modulation order )[0] # Channel
transmission and noise. No = pow(10., -esno / 10.) rx_tensor = channel(tx_tensor, No)
rx_tensor = np.array(rx_tensor) # Save the sample. rx_iq_data_filename =
"rx_iq_{}_esno{}_speed{}_ds{}_mcs{}_{}.pkl".format(channel_model, esno, speed,
delay_spread, mcs, sample) rx_iq_data_fullpath = os.path.join(dataset_dir,
rx_iq_data_filename) save_pickle(data=rx_tensor, filename=rx_iq_data_fullpath) #
Save noise power and SNR data as user data. user_data_filename =
"user_data_{}_esno{}_speed{}_ds{}_mcs{}_{}.pkl".format(channel_model, esno,
speed, delay_spread, mcs, sample) user_data_fullpath = os.path.join(dataset_dir,
user_data_filename) user_data = dict( snr=esno, noise_var=No )
save_pickle(data=user_data, filename=user_data_fullpath) pusch_record =
PuschRecord( # SCF FAPI 10.02 UL_TTI.request message parameters: pduIdx=0, SFN=
(sample // num_slots_per_frame) % 1023, Slot=slot_number, nPDUs=1,
RachPresent=0, nULSCH=1, nULCCH=0, nGroup=1, PDUSize=0, pduBitmap=1,
RNTI=rnti, Handle=0, BWPSize=273, BWPStart=0, SubcarrierSpacing=mu,
CyclicPrefix=0, targetCodeRate=coderate * 10, qamModOrder=mod_order,
mcsIndex=mcs, mcsTable=mcs_table - 1, # Different indexing TransformPrecoding=1,
# Disabled. dataScramblingId=data_scid, nrOfLayers=1,
ulDmrsSymbPos=dmrs_bit_array_to_fapi(dmrs_position), dmrsConfigType=0,
ulDmrsScramblingId=cell_id, puschIdentity=cell_id, SCID=scid,
numDmrsCdmGrpsNoData=2, dmrsPorts=1, # Note that FAPI uses a different format
compared to cuPHY. resourceAlloc=1, rbBitmap=np.array(36 * [0]), rbStart=0,
rbSize=273, VRBtoPRBMapping=0, FrequencyHopping=0,
txDirectCurrentLocation=0, uplinkFrequencyShift7p5khz=0,
StartSymbolIndex=start_sym, NrOfSymbols=num_symbols, puschData=None,
puschUci=None, puschPtrs=None, dftsOfdm=None, Beamforming=None, # SCF FAPI
10.02 RxData.indication message parameters: HarqID=0, PDULen=len(tb_input),
UL_CQI=255, # Set to invalid 0xFF. TimingAdvance=0, RSSI=65535, # Set to invalid
0xFFFF. macPdu=tb_input, TbCrcStatus=0, NumCb=0, CbCrcStatus=None,
rx_iq_data_filename=rx_iq_data_filename, user_data_filename=user_data_filename,
errInd = "" ) pusch_records.append(pusch_record) print("Saving...") df_filename =
os.path.join(dataset_dir, "l2_metadata.parquet") df =
pd.DataFrame.from_records(pusch_records, columns=PuschRecord._fields)
df.to_parquet(df_filename, engine="pyarrow") print("All done!")
```

```
Saving... All done!
```