Discover_Sionna

June 3, 2022

0.1 Discover Sionna

This example notebook will guide you through the basic principles and illustrates the key features of Sionna. With only a few commands, you can simulate the PHY-layer link-level performance for many 5G-compliant components, including easy visualization of the results.

0.1.1 Load Required Packages

The Sionna python package must be installed.

[31]:
```
import platform
import tensorflow as tf
import sionna

print(platform.version())
print(platform.python_version())
print(tf.__version__)
print(sionna.__version__)
tf.config.list_physical_devices()
```

```
Darwin Kernel Version 21.4.0: Mon Feb 21 20:36:53 PST 2022;
root:xnu-8020.101.4~2/RELEASE_ARM64_T8101
3.9.10
2.5.0
0.9.1
```

[31]:
```
[PhysicalDevice(name='/physical_device:CPU:0', device_type='CPU'),
 PhysicalDevice(name='/physical_device:GPU:0', device_type='GPU')]
```

[1]:
```
import numpy as np
import tensorflow as tf

# Import Sionna
try:
    import sionna
except ImportError as e:
    # Install Sionna if package is not already installed
    import os
    os.system("pip install sionna")
```

```
    import sionna

# IPython "magic function" for inline plots
%matplotlib inline
import matplotlib.pyplot as plt
```

```
Init Plugin
Init Graph Optimizer
Init Kernel
```

Tip: you can run bash commands in Jupyter via the `!` operator.

[2]:
```
!nvidia-smi
```

```
zsh:1: command not found: nvidia-smi
```

In case multiple GPUs are available, we restrict this notebook to single-GPU usage. You can ignore this command if only one GPU is available.

Further, we want to avoid that this notebook instantiates the whole GPU memory when initialized and set `memory_growth` as active.

Remark: Sionna does not require a GPU. Everything can also run on your CPU - but you may need to wait a little longer.

[3]:
```
# 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')
print('Number of GPUs available :', len(gpus))
if gpus:
    gpu_num = 0 # Index of the GPU to be used
    try:
        #tf.config.set_visible_devices([], 'GPU')
        tf.config.set_visible_devices(gpus[gpu_num], 'GPU')
        print('Only GPU number', gpu_num, 'used.')
        tf.config.experimental.set_memory_growth(gpus[gpu_num], True)
    except RuntimeError as e:
        print(e)
```

```
Number of GPUs available : 1
Only GPU number 0 used.
```

0.1.2 Sionna Data-flow and Design Paradigms

Sionna inherently parallelizes simulations via *batching*, i.e., each element in the batch dimension is simulated independently.

This means the first tensor dimension is always used for *inter-frame* parallelization similar to an outer *for-loop* in Matlab/NumPy simulations.

To keep the dataflow efficient, Sionna follows a few simple design principles:

- Signal-processing components are implemented as an individual Keras layer.
- `tf.float32` is used as preferred datatype and `tf.complex64` for complex-valued datatypes, respectively.
  This allows simpler re-use of components (e.g., the same scrambling layer can be used for binary inputs and LLR-values).
- Models can be developed in *eager mode* allowing simple (and fast) modification of system parameters.
- Number crunching simulations can be executed in the faster *graph mode* or even *XLA* acceleration is available for most components.
- Whenever possible, components are automatically differentiable via auto-grad to simplify the deep learning design-flow.
- Code is structured into sub-packages for different tasks such as channel coding, mapping,... (see API documentation for details).

The division into individual blocks simplifies deployment and all layers and functions comes with unittests to ensure their correct behavior.

These paradigms simplify the re-useability and reliability of our components for a wide range of communications related applications.

0.1.3 Let's Get Started - The First Layers (*Eager Mode*)

Every layer needs to be initialized once before it can be used.

Tip: use the API documentation to find an overview of all existing components.

We now want to transmit some symbols over an AWGN channel. First, we need to initialize the corresponding layer.

[4]:
```
channel = sionna.channel.AWGN() # init AWGN channel layer
```

In this first example, we want to add Gaussian noise to some given values of `x`.

Remember - the first dimension is the *batch-dimension*.

We simulate 2 message frames each containing 4 symbols.

Remark: the AWGN channel is defined to be complex-valued.

[5]:
```
# define a (complex-valued) tensor to be transmitted
x = tf.constant([[0., 1.5, 1., 0.],[-1., 0., -2, 3 ]], dtype=tf.complex64)

# let's have look at the shape
print("Shape of x: ", x.shape)
print("Values of x: ", x)
```

```
Metal device set to: Apple M1
Shape of x:  (2, 4)
Values of x:  tf.Tensor(
[[ 0. +0.j  1.5+0.j  1. +0.j  0. +0.j]
 [-1. +0.j  0. +0.j -2. +0.j  3. +0.j]], shape=(2, 4), dtype=complex64)
```

```
2022-06-03 06:53:46.603599: I
tensorflow/core/common_runtime/pluggable_device/pluggable_device_factory.cc:305]
Could not identify NUMA node of platform GPU ID 0, defaulting to 0. Your kernel
may not have been built with NUMA support.
2022-06-03 06:53:46.603790: I
tensorflow/core/common_runtime/pluggable_device/pluggable_device_factory.cc:271]
Created TensorFlow device (/job:localhost/replica:0/task:0/device:GPU:0 with 0
MB memory) -> physical PluggableDevice (device: 0, name: METAL, pci bus id:
<undefined>)
```

We want to simulate the channel at an SNR of 5 dB. For this, we can simply *call* the previously defined layer `channel`.

If you have never used Keras you can think of a layer as of a function: it has an input and returns the processed output.

Remark: Each time this cell is executed a new noise realization is drawn.

[6]:
```
ebno_db = 5

# calculate noise variance from given EbNo
no = sionna.utils.ebnodb2no(ebno_db = ebno_db,
                            num_bits_per_symbol=2, # QPSK
                            coderate=1)
y = channel([x, no])

print("Noisy symbols are: ", y)
```

```
Noisy symbols are:  tf.Tensor(
[[-0.1584535 -0.11372504j  1.7307588 -0.2803235j   0.95611054+0.12408835j
  -0.81951505-0.16318293j]
 [-0.5911193 +0.2608377j   0.07458896+0.08567969j -2.3748727 +0.3428662j
   2.9116876 +0.24346611j]], shape=(2, 4), dtype=complex64)
```

0.1.4 Batches and Multi-dimensional Tensors

Sionna natively supports multi-dimensional tensors.

Most layers operate at the last dimension and can have arbitrary input shapes (preserved at output).

Let us assume we want to add a CRC-24 check to 64 codewords of length 500 (e.g., different CRC per sub-carrier). Further, we want to parallelize the simulation over a batch of 100 samples.

[7]:
```
batch_size = 100 # outer level of parallelism
num_codewords = 64 # codewords per batch sample
info_bit_length = 500 # info bits PER codeword

source = sionna.utils.BinarySource() # yields random bits

u = source([batch_size, num_codewords, info_bit_length]) # call the source layer
print("Shape of u: ", u.shape)
```

```
# initialize an CRC encoder with the standard compliant "CRC24A" polynomial
encoder_crc = sionna.fec.crc.CRCEncoder("CRC24A")
decoder_crc = sionna.fec.crc.CRCDecoder(encoder_crc) # connect to encoder

# add the CRC to the information bits u
c = encoder_crc(u) # returns a list [c, crc_valid]
print("Shape of c: ", c.shape)
print("Processed bits: ", np.size(c.numpy()))

# we can also verify the results
# returns list of [info bits without CRC bits, indicator if CRC holds]
u_hat, crc_valid = decoder_crc(c)
print("Shape of u_hat: ", u_hat.shape)
print("Shape of crc_valid: ", crc_valid.shape)

print("Valid CRC check of first codeword: ", crc_valid.numpy()[0,0,0])
```

```
Shape of u:  (100, 64, 500)
Shape of c:  (100, 64, 524)
Processed bits:  3353600
Shape of u_hat:  (100, 64, 500)
Shape of crc_valid:  (100, 64, 1)
Valid CRC check of first codeword:  True
```

We want to do another simulation but for 5 independent users.

Instead of defining 5 different tensors, we can simply add another dimension.

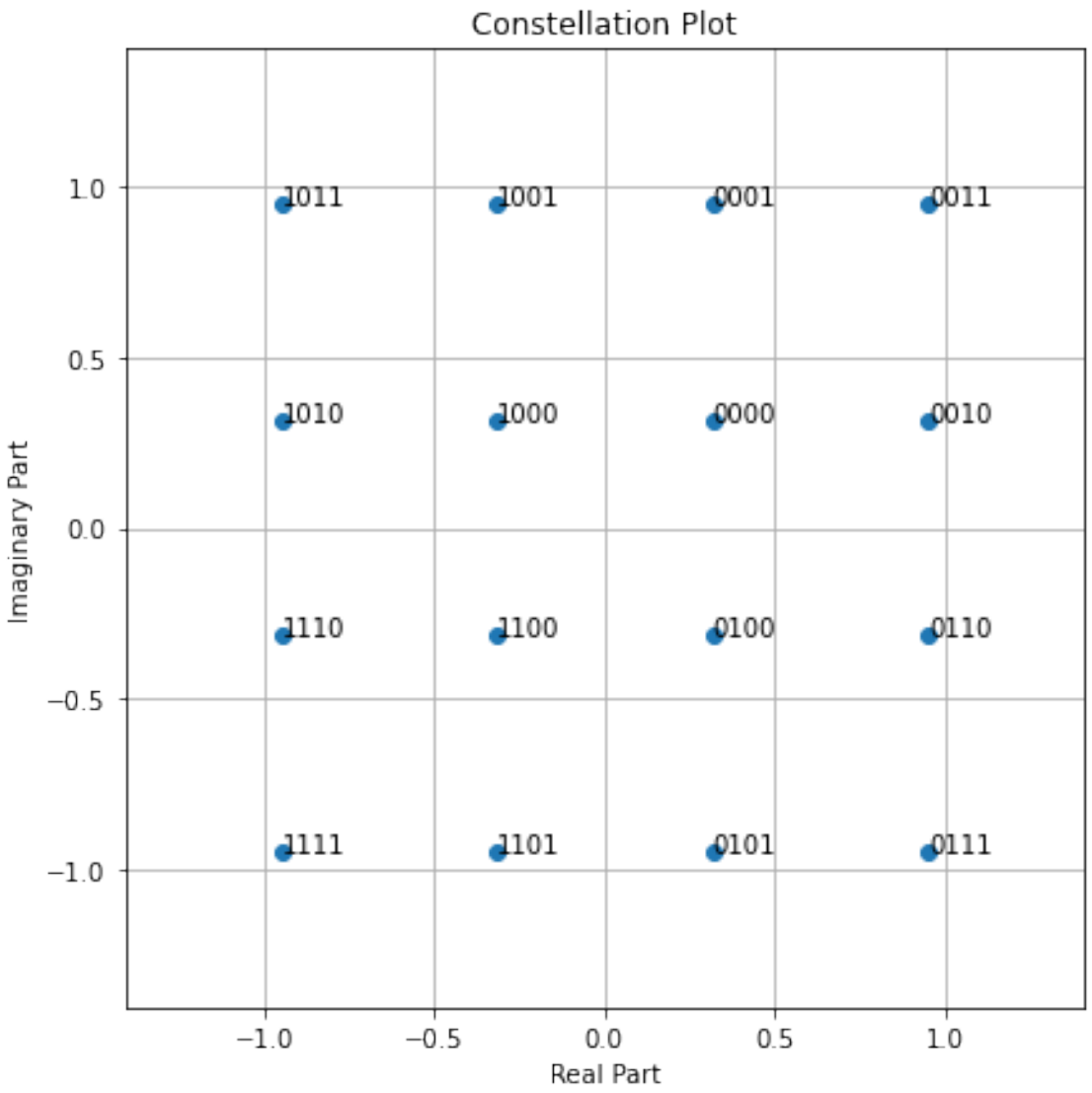
[8]:
```
num_users = 5

u = source([batch_size, num_users, num_codewords, info_bit_length])
print("New shape of u: ", u.shape)

# We can re-use the same encoder as before
c = encoder_crc(u)
print("New shape of c: ", c.shape)
print("Processed bits: ", np.size(c.numpy()))
```

```
New shape of u:  (100, 5, 64, 500)
New shape of c:  (100, 5, 64, 524)
Processed bits:  16768000
```

Often a good visualization of results helps to get new research ideas. Thus, Sionna has built-in plotting functions.

Let's have look at a 16-QAM constellation.

[9]:
```
constellation = sionna.mapping.Constellation("qam", num_bits_per_symbol=4)
constellation.show();
```



0.1.5 First Link-level Simulation

We can already build powerful code with a few simple commands.

As mentioned earlier, Sionna aims at hiding system complexity into Keras layers. However, we still want to provide as much flexibility as possible. Thus, most layers have several choices of init parameters, but often the default choice is a good start.

Tip: the API documentation provides many helpful references and implementation details.

[10]:
```
# system parameters
n_ldpc = 500 # LDPC codeword length
k_ldpc = 250 # number of info bits per LDPC codeword
coderate = k_ldpc / n_ldpc
```

```
num_bits_per_symbol = 4 # number of bits mapped to one symbol (cf. QAM)
```

Often, several different algorithms are implemented, e.g., the demapper supports *"true app"* demapping, but also *"max-log"* demapping.

The check-node (CN) update function of the LDPC BP decoder also supports multiple algorithms.

[11]:
```
demapping_method = "app" # try "max-log"
ldpc_cn_type = "boxplus" # try also "minsum"
```

Let us initialize all required components for the given system parameters.

[12]:
```
binary_source = sionna.utils.BinarySource()
encoder = sionna.fec.ldpc.encoding.LDPC5GEncoder(k_ldpc, n_ldpc)
constellation = sionna.mapping.Constellation("qam", num_bits_per_symbol)
mapper = sionna.mapping.Mapper(constellation=constellation)
channel = sionna.channel.AWGN()
demapper = sionna.mapping.Demapper(demapping_method,
                                   constellation=constellation)
decoder = sionna.fec.ldpc.decoding.LDPC5GDecoder(encoder,
                                                 hard_out=True,␣
 ↪cn_type=ldpc_cn_type,
                                                 num_iter=20)
```

We can now run the code in *eager mode*. This allows us to modify the structure at any time - you can try a different `batch_size` or a different SNR `ebno_db`.

[13]:
```
# simulation parameters
batch_size = 1000
ebno_db = 4

# Generate a batch of random bit vectors
b = binary_source([batch_size, k_ldpc])

# Encode the bits using 5G LDPC code
print("Shape before encoding: ", b.shape)
c = encoder(b)
print("Shape after encoding: ", c.shape)

# Map bits to constellation symbols
x = mapper(c)
print("Shape after mapping: ", x.shape)

# Transmit over an AWGN channel at SNR 'ebno_db'
no = sionna.utils.ebnodb2no(ebno_db, num_bits_per_symbol, coderate)
y = channel([x, no])
print("Shape after channel: ", y.shape)
```

```
# Demap to LLRs
llr = demapper([y, no])
print("Shape after demapping: ", llr.shape)

# LDPC decoding using 20 BP iterations
b_hat = decoder(llr)
print("Shape after decoding: ", b_hat.shape)

# calculate BERs
c_hat = tf.cast(tf.less(0.0, llr), tf.float32) # hard-decided bits before dec.
ber_uncoded = sionna.utils.metrics.compute_ber(c, c_hat)

ber_coded = sionna.utils.metrics.compute_ber(b, b_hat)

print("BER uncoded = {:.3f} at EbNo = {:.1f} dB".format(ber_uncoded, ebno_db))
print("BER after decoding = {:.3f} at EbNo = {:.1f} dB".format(ber_coded,␣
 ↪ebno_db))
print("In total {} bits were simulated".format(np.size(b.numpy())))
```

```
Shape before encoding:  (1000, 250)
Shape after encoding:  (1000, 500)
Shape after mapping:  (1000, 125)
Shape after channel:  (1000, 125)
Shape after demapping:  (1000, 500)
Shape after decoding:  (1000, 250)
BER uncoded = 0.120 at EbNo = 4.0 dB
BER after decoding = 0.011 at EbNo = 4.0 dB
In total 250000 bits were simulated
```

Just to summarize: we have simulated the transmission of 250,000 bits including higher-order modulation and channel coding!

But we can go even faster with the *TF graph execution*!

0.1.6 Setting up the End-to-end Model

We now define a *Keras model* that is more convenient for training and Monte-Carlo simulations.

We simulate the transmission over a time-varying multi-path channel (the *TDL-A* model from 3GPP TR38.901). For this, OFDM and a *conventional* bit-interleaved coded modulation (BICM) scheme with higher order modulation is used. The information bits are protected by a 5G-compliant LDPC code.

Remark: Due to the large number of parameters, we define them as dictionary.

[14]:
```
class e2e_model(tf.keras.Model): # inherits from keras.model
    """Example model for end-to-end link-level simulations.

    Parameters
    ----------
```

```
    params: dict
        A dictionary defining the system parameters.

    Input
    -----
    batch_size: int or tf.int
        The batch_sizeused for the simulation.

    ebno_db: float or tf.float
        A float defining the simulation SNR.

    Output
    ------
    (b, b_hat):
        Tuple:

    b: tf.float32
        A tensor of shape `[batch_size, k]` containing the transmitted
        information bits.

    b_hat: tf.float32
        A tensor of shape `[batch_size, k]` containing the receiver's
        estimate of the transmitted information bits.
    """
    def __init__(self,
                 params):
        super().__init__()


        # Define an OFDM Resource Grid Object
        self.rg = sionna.ofdm.ResourceGrid(
                            num_ofdm_symbols=params["num_ofdm_symbols"],
                            fft_size=params["fft_size"],
                            subcarrier_spacing=params["subcarrier_spacing"],
                            num_tx=1,
                            num_streams_per_tx=1,
                            cyclic_prefix_length=params["cyclic_prefix_length"],
                            pilot_pattern="kronecker",
                           ␣
↪pilot_ofdm_symbol_indices=params["pilot_ofdm_symbol_indices"])

        # Create a Stream Management object
        self.sm = sionna.mimo.StreamManagement(rx_tx_association=np.
↪array([[1]]),
                                               num_streams_per_tx=1)

        self.coderate = params["coderate"]
```

```
        self.num_bits_per_symbol = params["num_bits_per_symbol"]
        self.n = int(self.rg.num_data_symbols*self.num_bits_per_symbol)
        self.k = int(self.n*coderate)

        # Init layers
        self.binary_source = sionna.utils.BinarySource()
        self.encoder = sionna.fec.ldpc.encoding.LDPC5GEncoder(self.k, self.n)
        self.interleaver = sionna.fec.interleaving.RowColumnInterleaver(
                                        row_depth=self.num_bits_per_symbol)
        self.deinterleaver = sionna.fec.interleaving.Deinterleaver(self.
↪interleaver)
        self.mapper = sionna.mapping.Mapper("qam", self.num_bits_per_symbol)
        self.rg_mapper = sionna.ofdm.ResourceGridMapper(self.rg)
        self.tdl = sionna.channel.tr38901.TDL(model="A",
                            delay_spread=params["delay_spread"],
                            carrier_frequency=params["carrier_frequency"],
                            min_speed=params["min_speed"],
                            max_speed=params["max_speed"])

        self.channel = sionna.channel.OFDMChannel(self.tdl, self.rg,␣
↪add_awgn=True, normalize_channel=True)
        self.ls_est = sionna.ofdm.LSChannelEstimator(self.rg,␣
↪interpolation_type="nn")
        self.lmmse_equ = sionna.ofdm.LMMSEEqualizer(self.rg, self.sm)
        self.demapper = sionna.mapping.Demapper(params["demapping_method"],
                                                "qam", self.num_bits_per_symbol)
        self.decoder = sionna.fec.ldpc.decoding.LDPC5GDecoder(self.encoder,
                                                    hard_out=True,
                                                    cn_type=params["cn_type"],
                                                    num_iter=params["bp_iter"])

        print("Number of pilots: {}".format(self.rg.num_pilot_symbols))
        print("Number of data symbols: {}".format(self.rg.num_data_symbols))
        print("Number of resource elements: {}".format(
                                    self.rg.num_resource_elements))

        print("Pilot overhead: {:.2f}%".format(
                                    self.rg.num_pilot_symbols /
                                    self.rg.num_resource_elements*100))

        print("Cyclic prefix overhead: {:.2f}%".format(
                                    params["cyclic_prefix_length"] /
                                    (params["cyclic_prefix_length"]
                                    +params["fft_size"])*100))

        print("Each frame contains {} information bits".format(self.k))
```

```
    def call(self, batch_size, ebno_db):

        # Generate a batch of random bit vectors
        # We need two dummy dimension representing the number of
        # transmitters and streams per transmitter, respectively.
        b = self.binary_source([batch_size, 1, 1, self.k])

        # Encode the bits using the all-zero dummy encoder
        c = self.encoder(b)

        # Interleave the bits before mapping (BICM)
        c_int = self.interleaver(c)

        # Map bits to constellation symbols
        s = self.mapper(c_int)

        # Map symbols onto OFDM ressource grid
        x_rg = self.rg_mapper(s)

        # Transmit over noisy multi-path channel
        no = sionna.utils.ebnodb2no(ebno_db, self.num_bits_per_symbol, self.
↪coderate, self.rg)
        y = self.channel([x_rg, no])

        # LS Channel estimation with nearest pilot interpolation
        h_hat, err_var = self.ls_est ([y, no])

        # LMMSE Equalization
        x_hat, no_eff = self.lmmse_equ([y, h_hat, err_var, no])

        # Demap to LLRs
        llr = self.demapper([x_hat, no_eff])

        # Deinterleave before decoding
        llr_int = self.deinterleaver(llr)

        # Decode
        b_hat = self.decoder(llr_int)

        # number of simulated bits
        nb_bits = batch_size*self.k

        # transmitted bits and the receiver's estimate after decoding
        return b, b_hat
```

Let us define the system parameters for our simulation as dictionary:

[15]:
```
sys_params = {
    # Channel
    "carrier_frequency" : 3.5e9,
    "delay_spread" : 100e-9,
    "min_speed" : 3,
    "max_speed" : 3,
    "tdl_model" : "A",

    # OFDM
    "fft_size" : 256,
    "subcarrier_spacing" : 30e3,
    "num_ofdm_symbols" : 14,
    "cyclic_prefix_length" : 16,
    "pilot_ofdm_symbol_indices" : [2, 11],

    # Code & Modulation
    "coderate" : 0.5,
    "num_bits_per_symbol" : 4,
    "demapping_method" : "app",
    "cn_type" : "boxplus",
    "bp_iter" : 20
}
```

...and initialize the model:

[16]:
```
model = e2e_model(sys_params)
```

```
---------------------------------------------------------------------------
NotFoundError                             Traceback (most recent call last)
Input In [16], in <cell line: 1>()
----> 1 model = e2e_model(sys_params)

Input In [14], in e2e_model.__init__(self, params)
     32 super().__init__()
     35 # Define an OFDM Resource Grid Object
---> 36 self.rg = sionna.ofdm.ResourceGrid(
     37                     num_ofdm_symbols=params["num_ofdm_symbols"],
     38                     fft_size=params["fft_size"],
     39                     subcarrier_spacing=params["subcarrier_spacing"],
     40                     num_tx=1,
     41                     num_streams_per_tx=1,
     42                     cyclic_prefix_length=params["cyclic_prefix_length"]
     43                     pilot_pattern="kronecker",
     44␣
  ↪                  pilot_ofdm_symbol_indices=params["pilot_ofdm_symbol_indic s"])
     46 # Create a Stream Management object
     47 self.sm = sionna.mimo.StreamManagement(rx_tx_association=np.array([[1]] ,
     48                                        num_streams_per_tx=1)
```

```
File /opt/homebrew/Caskroom/miniforge/base/envs/env_tf/lib/python3.9/
 ↪site-packages/sionna/ofdm/resource_grid.py:85, in ResourceGrid.__init__(self,
 ↪num_ofdm_symbols, fft_size, subcarrier_spacing, num_tx, num_streams_per_tx,␣
 ↪cyclic_prefix_length, num_guard_carriers, dc_null, pilot_pattern,␣
 ↪pilot_ofdm_symbol_indices, dtype)
     83 self._dc_null = dc_null
     84 self._pilot_ofdm_symbol_indices = pilot_ofdm_symbol_indices
---> 85 self.pilot_pattern = pilot_pattern
     86 self._check_settings()

File /opt/homebrew/Caskroom/miniforge/base/envs/env_tf/lib/python3.9/
 ↪site-packages/sionna/ofdm/resource_grid.py:220, in ResourceGrid.
 ↪pilot_pattern(self, value)
    217     elif value=="kronecker":
    218         assert self._pilot_ofdm_symbol_indices is not None,\
    219             "You must provide pilot_ofdm_symbol_indices."
--> 220         value = KroneckerPilotPattern(self,
    221                 self._pilot_ofdm_symbol_indices, dtype=self._dtype)
    222 else:
    223     raise ValueError("Unsupported pilot_pattern")

File /opt/homebrew/Caskroom/miniforge/base/envs/env_tf/lib/python3.9/
 ↪site-packages/sionna/ofdm/pilot_pattern.py:349, in KroneckerPilotPattern.
 ↪__init__(self, resource_grid, pilot_ofdm_symbol_indices, normalize, seed,␣
 ↪dtype)
    346 for i in range(num_tx):
    347     for j in range(num_streams_per_tx):
    348         # Generate random QPSK symbols
--> 349         p = qam_source([1,1,num_pilot_symbols,num_pilots_per_symbol])
    351         # Place pilots spaced by num_seq to avoid overlap
    352         pilots[i,j,:,i*num_streams_per_tx+j::num_seq] = p

File /opt/homebrew/Caskroom/miniforge/base/envs/env_tf/lib/python3.9/
 ↪site-packages/tensorflow/python/keras/engine/base_layer.py:1030, in Layer.
 ↪__call__(self, *args, **kwargs)
   1026   inputs = self._maybe_cast_inputs(inputs, input_list)
   1028 with autocast_variable.enable_auto_cast_variables(
   1029     self._compute_dtype_object):
-> 1030   outputs = call_fn(inputs, *args, **kwargs)
   1032 if self._activity_regularizer:
   1033   self._handle_activity_regularization(inputs, outputs)

File /opt/homebrew/Caskroom/miniforge/base/envs/env_tf/lib/python3.9/
 ↪site-packages/sionna/utils/misc.py:211, in QAMSource.call(self, inputs)
    209 def call(self, inputs):
    210     num_points = 2**self._num_bits_per_symbol
--> 211     ind = self._rng.uniform(inputs, 0, num_points, tf.int32)
    212     x = tf.gather(self._constellation.points, ind)
```

```
    214     return x

File /opt/homebrew/Caskroom/miniforge/base/envs/env_tf/lib/python3.9/
 ↪site-packages/tensorflow/python/ops/stateful_random_ops.py:800, in Generator.
 ↪uniform(self, shape, minval, maxval, dtype, name)
    798 if dtype.is_integer:
    799   if compat.forward_compatible(2020, 10, 25):
--> 800     key, counter = self._prepare_key_counter(shape)
    801     return gen_stateless_random_ops_v2.stateless_random_uniform_int_v2(
    802         shape=shape,
    803         key=key,
   (…)
    807         alg=self.algorithm,
    808         name=name)
    809   return gen_stateful_random_ops.stateful_uniform_int(
    810       self.state.handle, self.algorithm, shape=shape,
    811       minval=minval, maxval=maxval, name=name)

File /opt/homebrew/Caskroom/miniforge/base/envs/env_tf/lib/python3.9/
 ↪site-packages/tensorflow/python/ops/stateful_random_ops.py:634, in Generator.
 ↪_prepare_key_counter(self, shape)
    632 def _prepare_key_counter(self, shape):
    633   delta = math_ops.reduce_prod(shape)
--> 634   counter_key = self.skip(delta)
    635   counter_size = _get_counter_size(self.algorithm)
    636   counter = array_ops.bitcast(counter_key[:counter_size], dtypes.uint64

File /opt/homebrew/Caskroom/miniforge/base/envs/env_tf/lib/python3.9/
 ↪site-packages/tensorflow/python/ops/stateful_random_ops.py:586, in Generator.
 ↪skip(self, delta)
    577 """Advance the counter of a counter-based RNG.
    578
    579 Args:
   (…)
    583     counter is an unspecified implementation detail.
    584 """
    585 if compat.forward_compatible(2020, 10, 25):
--> 586   return self._skip(delta)
    587 gen_stateful_random_ops.rng_skip(
    588     self.state.handle, math_ops.cast(self.algorithm, dtypes.int64),
    589     math_ops.cast(delta, dtypes.int64))

File /opt/homebrew/Caskroom/miniforge/base/envs/env_tf/lib/python3.9/
 ↪site-packages/tensorflow/python/ops/stateful_random_ops.py:616, in Generator.
 ↪_skip(self, delta)
    613       # In cross-replica context we need to use strategy.extended.updat
    614       return ds_context.get_strategy().extended.update(
    615           self.state, update_fn)
--> 616 return update_fn(self.state)
```

```
File /opt/homebrew/Caskroom/miniforge/base/envs/env_tf/lib/python3.9/
 ↪site-packages/tensorflow/python/ops/stateful_random_ops.py:600, in Generator.
 ↪_skip.<locals>.update_fn(v)
    599 def update_fn(v):
--> 600   return self._skip_single_var(v, delta)

File /opt/homebrew/Caskroom/miniforge/base/envs/env_tf/lib/python3.9/
 ↪site-packages/tensorflow/python/ops/stateful_random_ops.py:594, in Generator.
 ↪_skip_single_var(self, var, delta)
    592 def _skip_single_var(self, var, delta):
    593   # TODO(wangpeng): Cache the cast algorithm instead of casting␣
 ↪everytime.
--> 594   return gen_stateful_random_ops.rng_read_and_skip(
    595       var.handle, alg=math_ops.cast(self.algorithm, dtypes.int32),
    596       delta=math_ops.cast(delta, dtypes.uint64))

File /opt/homebrew/Caskroom/miniforge/base/envs/env_tf/lib/python3.9/
 ↪site-packages/tensorflow/python/ops/gen_stateful_random_ops.py:115, in␣
 ↪rng_read_and_skip(resource, alg, delta, name)
    113   return _result
    114 except _core._NotOkStatusException as e:
--> 115   _ops.raise_from_not_ok_status(e, name)
    116 except _core._FallbackException:
    117   pass

File /opt/homebrew/Caskroom/miniforge/base/envs/env_tf/lib/python3.9/
 ↪site-packages/tensorflow/python/framework/ops.py:6897, in␣
 ↪raise_from_not_ok_status(e, name)
   6895 message = e.message + (" name: " + name if name is not None else "")
   6896 # pylint: disable=protected-access
-> 6897 six.raise_from(core._status_to_exception(e.code, message), None)

File <string>:3, in raise_from(value, from_value)

NotFoundError: No registered 'RngReadAndSkip' OpKernel for 'GPU' devices␣
 ↪compatible with node {{node RngReadAndSkip}}
        .  Registered:  device='CPU'
 [Op:RngReadAndSkip]
```

As before, we can simply *call* the model to simulate the BER for the given simulation parameters.

[17]:
```
#simulation parameters
ebno_db = 10
batch_size = 200

# and call the model
b, b_hat = model(batch_size, ebno_db)
```

```
ber = sionna.utils.metrics.compute_ber(b, b_hat)
nb_bits = np.size(b.numpy())

print("BER: {:.4} at Eb/No of {} dB and {} simulated bits".format(ber.numpy(),␣
↪ebno_db, nb_bits))
```

```
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
Input In [17], in <cell line: 6>()
      3 batch_size = 200
      5 # and call the model
----> 6 b, b_hat = model(batch_size, ebno_db)
      8 ber = sionna.utils.metrics.compute_ber(b, b_hat)
      9 nb_bits = np.size(b.numpy())

NameError: name 'model' is not defined
```

0.1.7 Run some Throughput Tests (Graph Mode)

Sionna is not just an easy-to-use library, but also incredibly fast. Let us measure the throughput of the model defined above.

We compare *eager* and *graph* execution modes (see Tensorflow Doc for details), as well as *eager with XLA* (see https://www.tensorflow.org/xla#enable_xla_for_tensorflow_models). Note that we need to activate the sionna.config.xla_compat feature for XLA to work.

Tip: change the `batch_size` to see how the batch parallelism enhances the throughput. Depending on your machine, the `batch_size` may be too large.

[18]:
```
import time # this block requires the timeit library

batch_size = 200
ebno_db = 5 # evalaute SNR point
repetitions = 4 # throughput is averaged over multiple runs

def get_throughput(batch_size, ebno_db, model, repetitions=1):
    """ Simulate throughput in bit/s per ebno_db point.

    The results are average over `repetition` trials.

    Input
    -----
    batch_size: int or tf.int32
        Batch-size for evaluation.

    ebno_db: float or tf.float32
```

```
        A tensor containing the SNR points be evaluated

    model:
        Function or model that yields the transmitted bits `u` and the
        receiver's estimate `u_hat` for a given ``batch_size`` and
        ``ebno_db``.

    repetitions: int
        An integer defining how many trails of the throughput
        simulation are averaged.

    """


    # call model once to be sure it is compile properly
    # otherwise time to build graph is measured as well.
    u, u_hat = model(tf.constant(batch_size, tf.int32),
                     tf.constant(ebno_db, tf.float32))

    t_start = time.perf_counter()
    # average over multiple runs
    for _ in range(repetitions):
        u, u_hat = model(tf.constant(batch_size, tf.int32),
                         tf.constant(ebno_db, tf. float32))
    t_stop = time.perf_counter()

    # throughput in bit/s
    throughput = np.size(u.numpy())*repetitions / (t_stop - t_start)

    return throughput

# eager mode - just call the model
def run_eager(batch_size, ebno_db):
    return model(batch_size, ebno_db)

time_eager = get_throughput(batch_size, ebno_db, run_eager, repetitions=4)

# the decorator "@tf.function" enables the graph mode
@tf.function
def run_graph(batch_size, ebno_db):
    return model(batch_size, ebno_db)

time_graph = get_throughput(batch_size, ebno_db, run_graph, repetitions=4)

# the decorator "@tf.function(jit_compile=True)" enables the graph mode with XLA
# we need to activate the sionna.config.xla_compat feature for this to work
sionna.config.xla_compat=True
```

```
@tf.function(jit_compile=True)
def run_graph_xla(batch_size, ebno_db):
    return model(batch_size, ebno_db)

time_graph_xla = get_throughput(batch_size, ebno_db, run_graph_xla,␣
 ↪repetitions=4)
# we deactivate the sionna.config.xla_compat so that the cell can be run␣
 ↪mutiple times
sionna.config.xla_compat=False

print(f"Throughput in eager execution: {time_eager/1e6:.2f} Mb/s")
print(f"Throughput in graph execution: {time_graph/1e6:.2f} Mb/s")
print(f"Throughput in graph execution with XLA: {time_graph_xla/1e6:.2f} Mb/s")
```

```
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
Input In [18], in <cell line: 53>()
     50 def run_eager(batch_size, ebno_db):
     51     return model(batch_size, ebno_db)
---> 53 time_eager =␣
 ↪get_throughput(batch_size, ebno_db, run_eager, repetitions=4)
     55 # the decorator "@tf.function" enables the graph mode
     56 @tf.function
     57 def run_graph(batch_size, ebno_db):

Input In [18], in get_throughput(batch_size, ebno_db, model, repetitions)
      8 """ Simulate throughput in bit/s per ebno_db point.
      9
     10 The results are average over `repetition` trials.
   (…)
     28
     29 """
     32 # call model once to be sure it is compile properly
     33 # otherwise time to build graph is measured as well.
---> 34 u, u_hat = model(tf.constant(batch_size, tf.int32),
     35                  tf.constant(ebno_db, tf.float32))
     37 t_start = time.perf_counter()
     38 # average over multiple runs

Input In [18], in run_eager(batch_size, ebno_db)
     50 def run_eager(batch_size, ebno_db):
---> 51     return model(batch_size, ebno_db)

NameError: name 'model' is not defined
```

Obviously, *graph* execution (with XLA) yields much higher throughputs (at least if a fast GPU is

available). Thus, for exhaustive training and Monte-Carlo simulations the *graph* mode (with XLA and GPU acceleration) is the preferred choice.

0.1.8 Bit-Error Rate (BER) Monte-Carlo Simulations

Monte-Carlo simulations are omnipresent in todays communications research and development. Due its performant implementation, Sionna can be directly used to simulate BER at a performance that competes with compiled languages – but still keeps the flexibility of a script language.

[19]:
```
ebno_dbs = np.arange(0, 15, 1.)
batch_size = 200 # reduce in case you receive an out-of-memory (OOM) error

max_mc_iter = 1000 # max number of Monte-Carlo iterations before going to next␣
 ↪SNR point
num_target_block_errors = 500 # continue with next SNR point after target␣
 ↪number of block errors

# we use the built-in ber simulator function from Sionna which uses and early␣
 ↪stop after reaching num_target_errors
sionna.config.xla_compat=True
ber_mc,_ = sionna.utils.sim_ber(run_graph_xla, # you can also evaluate the␣
 ↪model directly
                                ebno_dbs,
                                batch_size=batch_size,
                                num_target_block_errors=num_target_block_errors,
                                max_mc_iter=max_mc_iter,
                                verbose=True) # print status and summary
sionna.config.xla_compat=False
```

```
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
Input In [19], in <cell line: 9>()
      7 # we use the built-in ber simulator function from Sionna which uses and␣
 ↪early stop after reaching num_target_errors
      8 sionna.config.xla_compat=True
----> 9 ber_mc,_ = sionna.utils.sim_ber(run_graph_xla, # you can also evaluate␣
 ↪the model directly
     10                                 ebno_dbs,
     11                                 batch_size=batch_size,
     12                                ␣
 ↪num_target_block_errors=num_target_block_errors,
     13                                 max_mc_iter=max_mc_iter,
     14                                 verbose=True) # print status and summar
     15 sionna.config.xla_compat=False

NameError: name 'run_graph_xla' is not defined
```

Let's look at the results.

[20]:
```
sionna.utils.plotting.plot_ber(ebno_dbs,
                               ber_mc,
                               legend="E2E Model",
                               ylabel="Coded BER");
```

```
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
Input In [20], in <cell line: 1>()
      1 sionna.utils.plotting.plot_ber(ebno_dbs,
----> 2                                ber_mc,
      3                                legend="E2E Model",
      4                                ylabel="Coded BER")

NameError: name 'ber_mc' is not defined
```

0.1.9 Conclusion

We hope you are excited about Sionna - there is much more to be discovered:

- TensorBoard debugging available
- Scaling to multi-GPU simulation is simple
- See the available tutorials for more advanced examples.

And if something is still missing - the project is open-source: you can modify, add, and extend any component at any time.