Sionna implementation with PyTorch Model

In this notebook:

The model is written in Pytorch and rest of the code is the same as in the original code.

Approach:

The data that is coming in is in tensorflow because Sionna is built using TF. Since your model is in Pytorch, we use dl_pack to convert the tensors into pytorch before we input the data to the model. The model returns a pytorch tensor. We convert this into a tensorflow tensor because thats what we want for future computation. The idea is to extend sionna for transformers.

Problems:

1. Loss of gradient information could be an issue.
2. Matching the data type
3. Proper conversion of tensors.

Install and Import

```
import os
gpu_num = 0 # Use "" to use the CPU
os.environ["CUDA_VISIBLE_DEVICES"] = f"{gpu_num}"
os.environ['TF_CPP_MIN_LOG_LEVEL'] = '3'

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

# Import TensorFlow and NumPy
import tensorflow as tf
# tf.compat.v1.disable_eager_execution()
# Avoid warnings from TensorFlow
tf.get_logger().setLevel('ERROR')
import numpy as np

# For saving complex Python data structures efficiently
import pickle

# For plotting
%matplotlib inline
import matplotlib.pyplot as plt

# For the implementation of the neural receiver
from tensorflow.keras import Model
from tensorflow.keras.layers import Layer, Conv2D, LayerNormalization
from tensorflow.nn import relu
```

Initialization section

```
# Bit per channel use
NUM_BITS_PER_SYMBOL = 2 # QPSK

# Minimum value of Eb/N0 [dB] for simulations
EBN0_DB_MIN = -3.0

# Maximum value of Eb/N0 [dB] for simulations
EBN0_DB_MAX = 5.0

# How many examples are processed by Sionna in parallel
BATCH_SIZE = 128

# Coding rate
CODERATE = 0.5

# Define the number of UT and BS antennas
```

```python
NUM_UT = 1
NUM_BS = 1
NUM_UT_ANT = 1
NUM_BS_ANT = 2

# The number of transmitted streams is equal to the number of UT antennas
# in both uplink and downlink
NUM_STREAMS_PER_TX = NUM_UT_ANT

# Create an RX-TX association matrix.
# RX_TX_ASSOCIATION[i,j]=1 means that receiver i gets at least one stream
# from transmitter j. Depending on the transmission direction (uplink or downlink),
# the role of UT and BS can change.
# For example, considering a system with 2 RX and 4 TX, the RX-TX
# association matrix could be
# [ [1 , 1, 0, 0],
#   [0 , 0, 1, 1] ]
# which indicates that the RX 0 receives from TX 0 and 1, and RX 1 receives from
# TX 2 and 3.
#
# In this notebook, as we have only a single transmitter and receiver,
# the RX-TX association matrix is simply:
RX_TX_ASSOCIATION = np.array([[1]])

# Instantiate a StreamManagement object
# This determines which data streams are determined for which receiver.
# In this simple setup, this is fairly easy. However, it can get more involved
# for simulations with many transmitters and receivers.
STREAM_MANAGEMENT = sn.mimo.StreamManagement(RX_TX_ASSOCIATION, NUM_STREAMS_PER_TX)

RESOURCE_GRID = sn.ofdm.ResourceGrid( num_ofdm_symbols=14,
                                      fft_size=76,
                                      subcarrier_spacing=30e3,
                                      num_tx=NUM_UT,
                                      num_streams_per_tx=NUM_STREAMS_PER_TX,
                                      cyclic_prefix_length=6,
                                      pilot_pattern="kronecker",
                                      pilot_ofdm_symbol_indices=[2,11]
                                      )

# Carrier frequency in Hz.
CARRIER_FREQUENCY = 2.6e9

# Antenna setting
UT_ARRAY = sn.channel.tr38901.Antenna(  polarization="single",
                                        polarization_type="V",
                                        antenna_pattern="38.901",
                                        carrier_frequency=CARRIER_FREQUENCY)
BS_ARRAY = sn.channel.tr38901.AntennaArray( num_rows=1,
                                            num_cols=int(NUM_BS_ANT/2),
                                            polarization="dual",
                                            polarization_type="cross",
                                            antenna_pattern="38.901", # Try 'omni'
                                            carrier_frequency=CARRIER_FREQUENCY)

# Nominal delay spread in [s]. Please see the CDL documentation
# about how to choose this value.
DELAY_SPREAD = 100e-9

# The `direction` determines if the UT or BS is transmitting.
# In the `uplink`, the UT is transmitting.
DIRECTION = "uplink"

# Suitable values are ["A", "B", "C", "D", "E"]
CDL_MODEL = "C"

# UT speed [m/s]. BSs are always assumed to be fixed.
# The direction of travel will chosen randomly within the x-y plane.
SPEED = 10.0

# Configure a channel impulse reponse (CIR) generator for the CDL model.
CDL = sn.channel.tr38901.CDL(CDL_MODEL,
                             DELAY_SPREAD,
                             CARRIER_FREQUENCY,
                             UT_ARRAY,
                             BS_ARRAY,
                             DIRECTION,
                             min_speed=SPEED)
```

PyTorch Model

```
import torch
import torch.nn as nn
import torch.nn.functional as F

class ResidualBlock(nn.Module):
    def __init__(self):
        super(ResidualBlock, self).__init__()

        # Layer normalization over the last three dimensions: time, frequency, conv 'channels'
        self.layer_norm_1 = nn.LayerNorm(normalized_shape=(14, 76, 128))
        self.conv_1 = nn.Conv2d(in_channels=128, out_channels=128, kernel_size=3, padding=1)

        # Layer normalization over the last three dimensions
        self.layer_norm_2 = nn.LayerNorm(normalized_shape=(14, 76, 128))
        self.conv_2 = nn.Conv2d(in_channels=128, out_channels=128, kernel_size=3, padding=1)

    def forward(self, x):
        z = self.layer_norm_1(x.permute(0, 2, 3, 1)).permute(0, 3, 1, 2)  # Adjust axes for layer norm
        z = F.relu(z)
        z = self.conv_1(z)
        z = self.layer_norm_2(z.permute(0, 2, 3, 1)).permute(0, 3, 1, 2)
        z = F.relu(z)
        z = self.conv_2(z)
        z = z + x  # Skip connection
        return z


import torch
import torch.nn as nn
import torch.nn.functional as F

class NeuralReceiver(nn.Module):
    def __init__(self, num_bits_per_symbol):
        super(NeuralReceiver, self).__init__()

        # Input convolution
        self.input_conv = nn.Conv2d(in_channels=5, out_channels=128, kernel_size=3, padding='same')

        # Residual blocks
        self.res_block_1 = ResidualBlock()
        self.res_block_2 = ResidualBlock()
        self.res_block_3 = ResidualBlock()
        self.res_block_4 = ResidualBlock()

        # Output convolution
        self.output_conv = nn.Conv2d(in_channels=128, out_channels=num_bits_per_symbol, kernel_size=3, padding='same')

    def forward(self, inputs):
        y, no = inputs
        ###############
        print("Type of y inside receiver: ",type(y))
        print("Type of no inside receiver: ",type(no))
        # Convert TensorFlow tensor to DLPack
        no = tf.experimental.dlpack.to_dlpack(no)
        y = tf.experimental.dlpack.to_dlpack(y)
        # Convert DLPack tensor to PyTorch tensor
        no = torch.utils.dlpack.from_dlpack(no)
        y = torch.utils.dlpack.from_dlpack(y)
        ###############
        # Assuming a single receiver, remove the num_rx dimension
        y = y.squeeze(1)

        # Feeding the noise power in log10 scale helps with the performance
        no = torch.log10(no)

        # Stacking the real and imaginary components of the different antennas along the 'channel' dimension
        y = y.permute(0, 2, 3, 1)  # Putting antenna dimension last
        no = no.reshape(no.shape[0], 1 , 1, 1)
        no = no.repeat(1, y.shape[1], y.shape[2], 1)
        z = torch.cat([torch.real(y), torch.imag(y.type(torch.cfloat)), no], dim=-1).permute(0, 3, 1, 2)
        # Input conv
        z = self.input_conv(z)

        # Residual blocks
        z = self.res_block_1(z)
        z = self.res_block_2(z)
        z = self.res_block_3(z)
        z = self.res_block_4(z)

        # Output conv
        z = self.output_conv(z)
        z = z.permute(0, 2, 3, 1)

        # Reset to channel last
        # Reshape the input to fit what the resource grid demapper is expected
        z = z.unsqueeze(1)
        z = z.unsqueeze(1)
        print('Output type of z: ', type(z))
        return z
```

OFDMSystemNeuralReceiver

```
# import tensorflow as tf
# import torch
# import torch.utils.dlpack
# import tensorflow_datasets as tfds
# from tensorflow.keras import Model
# import torch.nn.functional as F
# import numpy as np

class OFDMSystemNeuralReceiver(Model): # Inherits from Keras Model

    def __init__(self, training):
        super().__init__() # Must call the Keras model initializer

        self.training = training

        n = int(RESOURCE_GRID.num_data_symbols*NUM_BITS_PER_SYMBOL) # Number of coded bits
        k = int(n*CODERATE) # Number of information bits
        self.k = k
        self.n = n

        # The binary source will create batches of information bits
        self.binary_source = sn.utils.BinarySource()

        # The encoder maps information bits to coded bits
        self.encoder = sn.fec.ldpc.LDPC5GEncoder(k, n)

        # The mapper maps blocks of information bits to constellation symbols
        self.mapper = sn.mapping.Mapper("qam", NUM_BITS_PER_SYMBOL)

        # The resource grid mapper maps symbols onto an OFDM resource grid
        self.rg_mapper = sn.ofdm.ResourceGridMapper(RESOURCE_GRID)

        # Frequency domain channel
        self.channel = sn.channel.OFDMChannel(CDL, RESOURCE_GRID, add_awgn=True, normalize_channel=True, return_channel=False)

        # Neural receiver
        self.neural_receiver = NeuralReceiver(NUM_BITS_PER_SYMBOL)

        # Used to extract data-carrying resource elements
        self.rg_demapper = sn.ofdm.ResourceGridDemapper(RESOURCE_GRID, STREAM_MANAGEMENT)

        # The decoder provides hard-decisions on the information bits
        self.decoder = sn.fec.ldpc.LDPC5GDecoder(self.encoder, hard_out=True)

        # Loss function
        self.bce = tf.keras.losses.BinaryCrossentropy(from_logits=True) # Loss function

    @tf.function # Graph execution to speed things up
    def __call__(self, batch_size, ebno_db):
        no = sn.utils.ebnodb2no(ebno_db, num_bits_per_symbol=NUM_BITS_PER_SYMBOL, coderate=CODERATE, resource_grid=RESOURCE_GRID)

        # The neural receiver is expected no to have shape [batch_size].
        if len(no.shape) == 0:
            no = tf.fill([batch_size], no)

        # Transmitter
        # Outer coding is only performed if not training
        if self.training:
            codewords = self.binary_source([batch_size, NUM_UT, NUM_UT_ANT, self.n])
        else:
            bits = self.binary_source([batch_size, NUM_UT, NUM_UT_ANT, self.k])
            codewords = self.encoder(bits)
        x = self.mapper(codewords)
        x_rg = self.rg_mapper(x)

        # Channel
        y = self.channel([x_rg, no])

        # Receiver
        print('y before the receiver: ', type(y))
        print('no before the receiver: ', type(no))
        llr = self.neural_receiver([y, no])
        llr = torch.utils.dlpack.to_dlpack(llr)
        llr = tf.experimental.dlpack.from_dlpack(llr)
        print('llr after the receiver: ', type(llr))

        llr = self.rg_demapper(llr) # Extract data-carrying resource elements. The other LLrs are discarded
        print('Type: ',type(llr))
        llr = tf.reshape(llr, [batch_size, NUM_UT, NUM_UT_ANT, self.n]) # Reshape the LLRs to fit what the outer decoder is expected
        if self.training:
            loss = self.bce(codewords, llr)
            return loss
        else:
            bits_hat = self.decoder(llr)
            return bits, bits_hat
```

Training section

```
# Set a seed for reproducibility
tf.random.set_seed(1)

# Number of iterations used for training
NUM_TRAINING_ITERATIONS = 50

# Instantiate the model for training
model = OFDMSystemNeuralReceiver(training=True)

# Adam optimizer (TensorFlow version)
optimizer = tf.keras.optimizers.Adam()

# Training loop
for i in range(NUM_TRAINING_ITERATIONS):
    # Sample a batch of SNRs.
    ebno_db = tf.random.uniform(shape=[BATCH_SIZE], minval=EBN0_DB_MIN, maxval=EBN0_DB_MAX)

    # Forward pass
    with tf.GradientTape() as tape:
        loss = model(BATCH_SIZE, ebno_db)

    # Compute and apply gradients
    weights = model.trainable_variables
    grads = tape.gradient(loss, weights)
    optimizer.apply_gradients(zip(grads, weights))

    # Print progress
    if i % 100 == 0:
        print(f"{i}/{NUM_TRAINING_ITERATIONS}  Loss: {loss:.2E}", end="\r")

# Save the weights in a file
weights = model.get_weights()
with open('weights-ofdm-neuralrx.pkl', 'wb') as f:
    pickle.dump(weights, f)

```

```
y before the receiver:  <class 'tensorflow.python.framework.ops.SymbolicTensor'>
no before the receiver:  <class 'tensorflow.python.framework.ops.SymbolicTensor'>
Type of y inside receiver:  <class 'tensorflow.python.framework.ops.SymbolicTensor'>
Type of no inside receiver:  <class 'tensorflow.python.framework.ops.SymbolicTensor'>
---------------------------------------------------------------------------
InvalidArgumentError                      Traceback (most recent call last)
<ipython-input-9-1fa9f459896a> in <cell line: 14>()
     18     # Forward pass
     19     with tf.GradientTape() as tape:
---> 20         loss = model(BATCH_SIZE, ebno_db)
     21 
     22     # Compute and apply gradients

                                    1 frames
/usr/local/lib/python3.10/dist-packages/tensorflow/python/eager/polymorphic_function/autograph_util.py in autograph_handler(*args,
**kwargs)
     50     except Exception as e:  # pylint:disable=broad-except
     51       if hasattr(e, "ag_error_metadata"):
---> 52         raise e.ag_error_metadata.to_exception(e)
     53       else:
     54         raise

InvalidArgumentError: in user code:

    File "<ipython-input-8-e60021f9d001>", line 72, in __call__  *
        llr = self.neural_receiver([y, no])
    File "/usr/local/lib/python3.10/dist-packages/torch/nn/modules/module.py", line 1553, in _wrapped_call_impl  *
        return self._call_impl(*args, **kwargs)
    File "/usr/local/lib/python3.10/dist-packages/torch/nn/modules/module.py", line 1562, in _call_impl  *
        return forward_call(*args, **kwargs)
    File "<ipython-input-4-4477d98104f1>", line 54, in forward  *
        no = tf.experimental.dlpack.to_dlpack(no)

    InvalidArgumentError: The argument to `to_dlpack` must be a TF tensor, not Python object
```

Next steps: Explain error