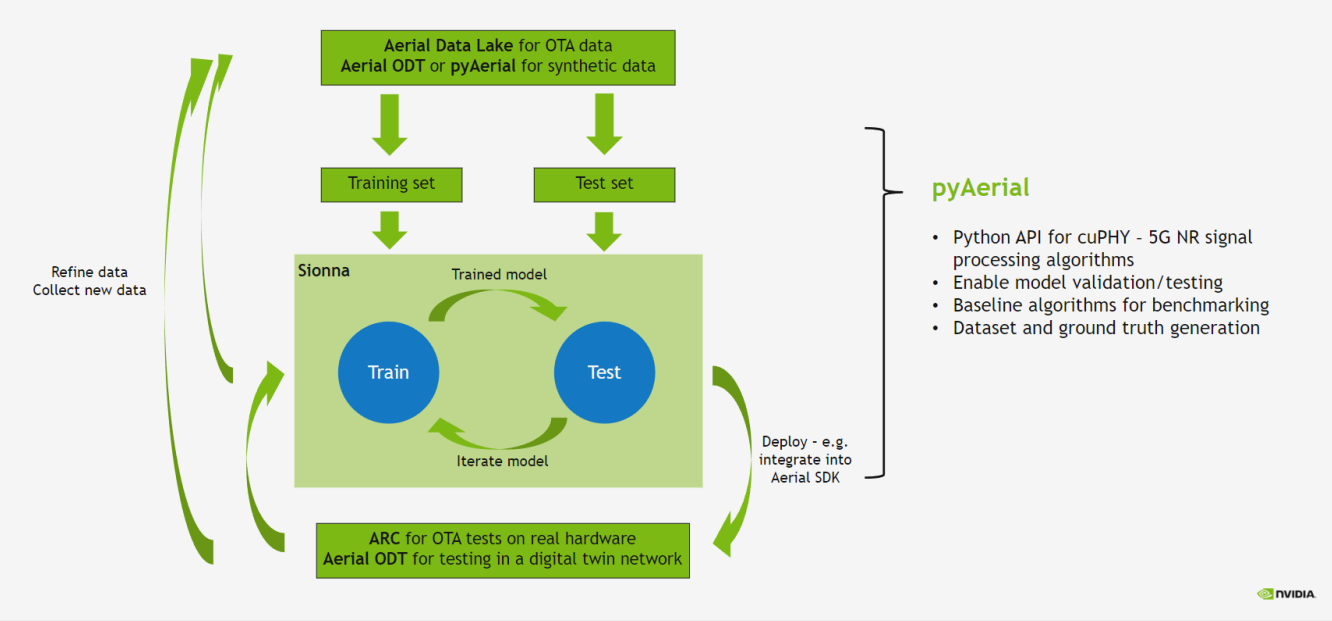
# LLRNet: Model training and testing

# Table of contents

# List of Figures

The wireless ML design flow using Aerial is depicted in the figure below.




In this notebook, we use the generated LLRNet data for training and validating LLRNet as part of the PUSCH receiver chain, implemented using pyAerial, with the Aerial cuPHY library working as the backend. The LLRNet is plugged in the PUSCH receiver chain in place of the conventional soft demapper. So this notebook works as an example of using pyAerial for model validation.

Finally, the model is exported into a format consumed by the TensorRT inference engine that is used for integrating the model into Aerial CUDA-Accelerated RAN for testing the model with real hardware in an over the air environment.

**Note 1:** This notebook requires that the Aerial test vectors have been generated. The test vector directory is set below in `AERIAL_TEST_VECTOR_DIR` variable. **Note 2:** This notebook also requires that the notebook example on LLRNet dataset generation has been run first.

# Imports

[1]:

```
%matplotlib inline import os os.environ["CUDA_VISIBLE_DEVICES"] = "0"
os.environ['TF_CPP_MIN_LOG_LEVEL'] = "3" # Silence TensorFlow. import cuda import
```

h5py as h5 import numpy as np import matplotlib.pyplot as plt import pandas as pd import tensorflow as tf from tensorflow import keras from tensorflow.keras import layers import tf2onnx import onnx from IPython.display import Markdown from IPython.display import display *# PyAerial components* from aerial.phy5g.algorithms import ChannelEstimator from aerial.phy5g.algorithms import ChannelEqualizer from aerial.phy5g.algorithms import NoiseIntfEstimator from aerial.phy5g.algorithms import Demapper from aerial.phy5g.ldpc import LdpcDeRateMatch from aerial.phy5g.ldpc import LdpcDecoder from aerial.phy5g.ldpc import code_block_desegment from aerial.util.cuda import get_cuda_stream from aerial.util.data import load_pickle from aerial.util.fapi import dmrs_fapi_to_bit_array *# 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)

[2]:

tb_errors = dict(aerial=dict(), llrnet=dict(), logmap=dict()) tb_count = dict(aerial=dict(), llrnet=dict(), logmap=dict())

[3]:

*# Dataset root directory.* DATA_DIR = "data/" *# Aerial test vector directory.* AERIAL_TEST_VECTOR_DIR = "/mnt/cicd_tvs/develop/GPU_test_input/" *# LLRNet dataset directory.* dataset_dir = DATA_DIR + "example_llrnet_dataset/QPSK/" *# Training vs. testing SNR. Assume these exist in the dataset.* train_snr = [-7.75, -7.5, -7.25, -7.0, -6.75, -6.5] test_snr = [-7.75, -7.5, -7.25, -7.0, -6.75, -6.5] *# Training, validation and test split in percentages if the same SNR is used for # training and testing.* train_split = 45 val_split = 5 test_split = 50 *# Training hyperparameters.* batch_size = 32 epochs = 5 step = tf.Variable(0, trainable=False) boundaries = [350000, 450000] values = [5e-4, 1e-4, 1e-5] *# values = [0.05, 0.01, 0.001]* learning_rate_fn = tf.keras.optimizers.schedules.PiecewiseConstantDecay(boundaries, values) optimizer = tf.keras.optimizers.Adam(learning_rate=learning_rate_fn, weight_decay=1e-4) *# optimizer = tf.keras.optimizers.experimental.SGD(learning_rate=0.05, weight_decay=1e-4,*

*momentum=0.9) # Modulation order. LLRNet needs to be trained separately for each modulation order.* mod_order = 2

## Define the LLRNet model

The LLRNet model follows the original paper

15. Shental, J. Hoydis, "*'Machine LLRning': Learning to Softly Demodulate*", https://arxiv.org/abs/1907.01512

and is a very simple MLP model with a single hidden layer. It takes the equalized symbols in its input with the real and imaginary parts separated, and outputs soft bits (log-likelihood ratios) that can be further fed into LDPC (de)rate matching and decoding.

[4]:

model = keras.Sequential( [ layers.Dense(16, input_dim=2, activation="relu"), layers.Dense(8, activation="linear") ] ) def loss(llr, predictions): mae = tf.abs(predictions[:, :mod_order] - llr) mse = tf.reduce_mean(tf.square(mae)) return mse

## Pre-process the dataset

Here, the dataset gets loaded and split into training, validation, and testing datasets, as well as put in the right format for the model.

[5]:

*# Load the main data file* try: df = pd.read_parquet(dataset_dir + "l2_metadata.parquet", engine="pyarrow") except FileNotFoundError: display(Markdown("**Data not found - has llrnet_dataset_generation.ipynb been run?**")) *# Query the entries for the selected modulation order.* df = df[df["qamModOrder"] == mod_order] *# Collect the dataset by SNR.* llrs = dict() eq_syms = dict() indices = dict() for pusch_record in df.itertuples(): user_data_filename = dataset_dir + pusch_record.user_data_filename user_data = load_pickle(user_data_filename) if user_data["snr"] not in llrs.keys():

```
llrs[user_data["snr"]] = [] eq_syms[user_data["snr"]] = [] indices[user_data["snr"]] = []
llrs[user_data["snr"]].append(user_data["map_llrs"])
eq_syms[user_data["snr"]].append(user_data["eq_syms"])
indices[user_data["snr"]].append(pusch_record.Index) llr_train, llr_val = [], []
sym_train, sym_val = [], [] test_indices = [] for key in llrs.keys(): llrs[key] =
np.stack(llrs[key]) eq_syms[key] = np.stack(eq_syms[key]) # Randomize the order.
permutation = np.arange(llrs[key].shape[0]) np.random.shuffle(permutation)
llrs[key] = llrs[key][permutation, ...] eq_syms[key] = eq_syms[key][permutation, ...]
indices[key] = list(np.array(indices[key])[permutation]) # Separate real and imaginary
parts of the symbols. eq_syms[key] = np.stack((np.real(eq_syms[key]),
np.imag(eq_syms[key]))) num_slots = llrs[key].shape[0] if key in train_snr and key in
test_snr: num_train_slots = int(np.round(train_split / 100 * num_slots))
num_val_slots = int(np.round(val_split / 100 * num_slots)) num_test_slots =
int(np.round(test_split / 100 * num_slots)) elif key in train_snr: num_train_slots =
int(np.round(train_split / (train_split + val_split) * num_slots)) num_val_slots =
int(np.round(val_split / (train_split + val_split) * num_slots)) num_test_slots = 0 elif
key in test_snr: num_train_slots = 0 num_val_slots = 0 num_test_slots = num_slots
else: num_train_slots = 0 num_val_slots = 0 num_test_slots = 0 # Collect
training/validation/testing sets. llr_train.append(llrs[key][:num_train_slots, ...])
llr_val.append(llrs[key][num_train_slots:num_train_slots+num_val_slots, ...])
sym_train.append(eq_syms[key][:, :num_train_slots, ...])
sym_val.append(eq_syms[key][:, num_train_slots:num_train_slots+num_val_slots,
...]) # Just indices for the test set. test_indices += indices[key]
[num_train_slots+num_val_slots:num_train_slots+num_val_slots+num_test_slots]
llr_train = np.transpose(np.concatenate(llr_train, axis=0), (1, 0, 2)) llr_val =
np.transpose(np.concatenate(llr_val, axis=0), (1, 0, 2)) sym_train =
np.concatenate(sym_train, axis=1) sym_val = np.concatenate(sym_val, axis=1) #
Fetch the total number of slots in each set. num_train_slots = llr_train.shape[1]
num_val_slots = llr_val.shape[1] num_test_slots = len(test_indices) normalizer = 1.0
#np.sqrt(np.var(llr_train)) llr_train = llr_train / normalizer llr_val = llr_val / normalizer #
Reshape into samples x mod_order array. llr_train = llr_train.reshape(mod_order, -1).T
llr_val = llr_val.reshape(mod_order, -1).T # Reshape into samples x 2 array. sym_train
= sym_train.reshape(2, -1).T sym_val = sym_val.reshape(2, -1).T print(f"Total number
of slots in the training set:{num_train_slots}") print(f"Total number of slots in the
validation set:{num_val_slots}") print(f"Total number of slots in the test set:
{num_test_slots}")
```

Total number of slots in the training set: 5400 Total number of slots in the validation set: 600 Total number of slots in the test set: 6000

## Model training and validation

Model training is done using Keras here.

[6]:

```
print("Training...") model.compile(loss=loss, optimizer=optimizer, metrics=[loss]) model.fit( x=sym_train, y=llr_train, batch_size=batch_size, epochs=epochs, verbose=1, validation_data=(sym_val, llr_val), shuffle=True )
```

```
Training... Epoch 1/5
```

```
WARNING: All log messages before absl::InitializeLog() is called are written to STDERR I0000 00:00:1712157396.498651 24812 device_compiler.h:186] Compiled cluster using XLA! This line is logged at most once for the lifetime of the process.
```

```
552825/552825 [==============================] - 765s 1ms/step - loss: 87.9453 - val_loss: 87.2937 Epoch 2/5 552825/552825 [==============================] - 769s 1ms/step - loss: 86.1241 - val_loss: 87.2001 Epoch 3/5 552825/552825 [==============================] - 771s 1ms/step - loss: 86.0340 - val_loss: 87.1084 Epoch 4/5 552825/552825 [==============================] - 769s 1ms/step - loss: 85.9486 - val_loss: 87.0212 Epoch 5/5 552825/552825 [==============================] - 768s 1ms/step - loss: 85.8626 - val_loss: 86.9364
```

[6]:

<keras.src.callbacks.History at 0x7f3e992f9db0>

## Define a PUSCH receiver chain using pyAerial

This class encapsulates the whole PUSCH receiver chain. The components include channel estimation, noise and interference estimation, channel equalization and soft demapping, LDPC (de)rate matching and LDPC decoding. The receiver outputs the received transport block in bytes.

The soft demapping part can be replaced by LLRNet.

[7]:

class Receiver: """PUSCH receiver class. This class encapsulates the whole PUSCH receiver chain built using pyAerial components. """ def __init__(self, num_rx_ant, enable_pusch_tdi, eq_coeff_algo): """Initialize the PUSCH receiver."""" self.cuda_stream = get_cuda_stream() *# Build the components of the receiver.* self.channel_estimator = ChannelEstimator( num_rx_ant=num_rx_ant, cuda_stream=self.cuda_stream ) self.channel_equalizer = ChannelEqualizer( num_rx_ant=num_rx_ant, enable_pusch_tdi=enable_pusch_tdi, eq_coeff_algo=eq_coeff_algo, cuda_stream=self.cuda_stream ) self.noise_intf_estimator = NoiseIntfEstimator( num_rx_ant=num_rx_ant, eq_coeff_algo=eq_coeff_algo, cuda_stream=self.cuda_stream ) self.demapper = Demapper(mod_order=mod_order) self.derate_match = LdpcDeRateMatch( enable_scrambling=True, cuda_stream=self.cuda_stream ) self.decoder = LdpcDecoder(cuda_stream=self.cuda_stream) self.llr_method = "llrnet" def set_llr_method(self, method): """Set the used LLR computation method. Args: method (str): Either "aerial" meaning the conventional log-likelihood ratio computation, or "llrnet" for using LLRNet instead. """ if method not in ["aerial", "logmap", "llrnet"]: raise ValueError("Invalid LLR computation method!") self.llr_method = method def run( self, rx_slot, num_ues, slot, num_dmrs_cdm_grps_no_data, dmrs_scrm_id, start_prb, num_prbs, dmrs_syms, dmrs_max_len, dmrs_add_ln_pos, start_sym, num_symbols, scids, layers,

dmrs_ports, rntis, data_scids, code_rates, mod_orders, tb_sizes, rvs, ndis): """Run the receiver.""" *# Channel estimation.* ch_est = self.channel_estimator.estimate( rx_slot=rx_slot, num_ues=num_ues, slot=slot, num_dmrs_cdm_grps_no_data=num_dmrs_cdm_grps_no_data, dmrs_scrm_id=dmrs_scrm_id, start_prb=start_prb, num_prbs=num_prbs, dmrs_syms=dmrs_syms, dmrs_max_len=dmrs_max_len, dmrs_add_ln_pos=dmrs_add_ln_pos, start_sym=start_sym, num_symbols=num_symbols, scids=scids, layers=layers, dmrs_ports=dmrs_ports ) *# Noise and interference estimation.* lw_inv, noise_var_pre_eq = self.noise_intf_estimator.estimate( rx_slot=rx_slot, channel_est=ch_est, num_ues=num_ues, slot=slot, num_dmrs_cdm_grps_no_data=num_dmrs_cdm_grps_no_data, dmrs_scrm_id=dmrs_scrm_id, start_prb=start_prb, num_prbs=num_prbs, dmrs_syms=dmrs_syms, dmrs_max_len=dmrs_max_len, dmrs_add_ln_pos=dmrs_add_ln_pos, start_sym=start_sym, num_symbols=num_symbols, scids=scids, layers=layers, dmrs_ports=dmrs_ports ) *# Channel equalization and soft demapping. Note that the cuPHY kernel actually computes both # the equalized symbols and the LLRs.* llr, eq_sym = self.channel_equalizer.equalize( rx_slot=rx_slot, channel_est=ch_est, lw_inv=lw_inv, noise_var_pre_eq=noise_var_pre_eq, num_ues=num_ues, num_dmrs_cdm_grps_no_data=num_dmrs_cdm_grps_no_data, start_prb=start_prb, num_prbs=num_prbs, dmrs_syms=dmrs_syms, dmrs_max_len=dmrs_max_len, dmrs_add_ln_pos=dmrs_add_ln_pos, start_sym=start_sym, num_symbols=num_symbols, layers=layers, mod_orders=mod_orders ) *# Use the LLRNet model here to get the log-likelihood ratios.* num_data_sym = (np.array(dmrs_syms[start_sym:start_sym + num_symbols]) == 0).sum() if self.llr_method == "llrnet": *# Put the input in the right format.* eq_sym_input = np.stack((np.real(eq_sym[0]), np.imag(eq_sym[0]))).reshape(2, -1).T *# Run the model.* llr_output = model(eq_sym_input) *# Reshape the output in the right format for the LDPC decoding process.* llr_output = np.array(llr_output)[..., :mod_orders[0]].T.reshape(mod_orders[0], layers[0], num_prbs * 12, num_data_sym) llr_output *= normalizer elif self.llr_method == "aerial": llr_output = llr[0] elif self.llr_method == "logmap": inv_noise_var_lin = self.channel_equalizer.ree_diag_inv[0] llr_output = self.demapper.demap(eq_sym[0], inv_noise_var_lin[..., None]) *# De-rate matching and descrambling.* cinit = (rntis[0] << 15) + data_scids[0] rate_match_len = num_data_sym * mod_orders[0] * num_prbs *

```
12 * layers[0] coded_blocks = self.derate_match.derate_match( input_data=llr_output, tb_size=tb_sizes[0], code_rate=code_rates[0], rate_match_len=rate_match_len, mod_order=mod_orders[0], num_layers=layers[0], redundancy_version=rvs[0], ndi=ndis[0], cinit=cinit ) # LDPC decoding of the derate matched blocks. code_blocks = self.decoder.decode( input_llr=coded_blocks, tb_size=tb_sizes[0], code_rate=code_rates[0], redundancy_version=rvs[0], rate_match_len=rate_match_len ) # Combine the code blocks into a transport block. tb = code_block_desegment( code_blocks=code_blocks, tb_size=tb_sizes[0], code_rate=code_rates[0], return_bits=False ) return tb
```

# Model testing on Aerial test vectors

[8]:

```
if mod_order == 2: test_vector_filename = "TVnr_7201_PUSCH_gNB_CUPHY_s0p0.h5" elif mod_order == 4: test_vector_filename = "TVnr_7916_PUSCH_gNB_CUPHY_s0p0.h5" elif mod_order == 6: test_vector_filename = "TVnr_7203_PUSCH_gNB_CUPHY_s0p0.h5" filename = AERIAL_TEST_VECTOR_DIR + test_vector_filename input_file = h5.File(filename, "r") num_rx_ant = input_file["gnb_pars"]["nRx"][0] enable_pusch_tdi = input_file["gnb_pars"]["TdiMode"][0] eq_coeff_algo = input_file["gnb_pars"]["eqCoeffAlgoIdx"][0] receiver = Receiver( num_rx_ant=num_rx_ant, enable_pusch_tdi=enable_pusch_tdi, eq_coeff_algo=eq_coeff_algo ) # Extract the test vector data and parameters. rx_slot = np.array(input_file["DataRx"])["re"] + 1j * np.array(input_file["DataRx"])["im"] rx_slot = rx_slot.transpose(2, 1, 0) num_ues = input_file["ueGrp_pars"]["nUes"][0] start_prb = input_file["ueGrp_pars"]["startPrb"][0] num_prbs = input_file["ueGrp_pars"]["nPrb"][0] start_sym = input_file["ueGrp_pars"]["StartSymbolIndex"][0] num_symbols = input_file["ueGrp_pars"]["NrOfSymbols"][0] dmrs_sym_loc_bmsk = input_file["ueGrp_pars"]["dmrsSymLocBmsk"][0] dmrs_scrm_id = input_file["tb_pars"]["dmrsScramId"][0] dmrs_max_len = input_file["tb_pars"]["dmrsMaxLength"][0] dmrs_add_ln_pos = input_file["tb_pars"]["dmrsAddlPosition"][0] num_dmrs_cdm_grps_no_data = input_file["tb_pars"]["numDmrsCdmGrpsNoData"][0] mod_orders = input_file["tb_pars"]["qamModOrder"] layers = input_file["tb_pars"]["numLayers"] scids = input_file["tb_pars"]["nSCID"] dmrs_ports = input_file["tb_pars"]["dmrsPortBmsk"]
```

```
slot = np.array(input_file["gnb_pars"]["slotNumber"])[0] tb_sizes = 8 * input_file["tb_pars"]["nTbByte"] code_rates = [input_file["tb_pars"]["targetCodeRate"][0] / 10240.] rvs = input_file["tb_pars"]["rv"] ndis = input_file["tb_pars"]["ndi"] rntis = input_file["tb_pars"]["nRnti"] data_scids = input_file["tb_pars"]["dataScramId"] dmrs_syms = dmrs_fapi_to_bit_array(dmrs_sym_loc_bmsk) # Run the receiver with the test vector parameters. receiver.set_llr_method("llrnet") tb = receiver.run( rx_slot=rx_slot, num_ues=num_ues, slot=slot, num_dmrs_cdm_grps_no_data=num_dmrs_cdm_grps_no_data, dmrs_scrm_id=dmrs_scrm_id, start_prb=start_prb, num_prbs=num_prbs, dmrs_syms=dmrs_syms, dmrs_max_len=dmrs_max_len, dmrs_add_ln_pos=dmrs_add_ln_pos, start_sym=start_sym, num_symbols=num_symbols, scids=scids, layers=layers, dmrs_ports=dmrs_ports, rntis=rntis, data_scids=data_scids, code_rates=code_rates, mod_orders=mod_orders, tb_sizes=tb_sizes, rvs=rvs, ndis=ndis ) # Check that the received TB matches with the transmitted one. if np.array_equal(np.array(input_file["tb_data"])[:tb_sizes[0]//8, 0], tb[:tb_sizes[0]//8]): print("CRC check passed!") else: print("CRC check failed!")
```

```
CRC check passed!
```

## Model testing on synthetic data

[9]:

```
for pusch_record in df.take(test_indices).itertuples(index=False): user_data_filename = dataset_dir + pusch_record.user_data_filename user_data = load_pickle(user_data_filename) snr = user_data["snr"] rx_iq_data_filename = dataset_dir + pusch_record.rx_iq_data_filename rx_slot = load_pickle(rx_iq_data_filename) num_ues = 1 start_prb = pusch_record.rbStart num_prbs = pusch_record.rbSize start_sym = pusch_record.StartSymbolIndex num_symbols = pusch_record.NrOfSymbols dmrs_syms = dmrs_fapi_to_bit_array(pusch_record.ulDmrsSymbPos) dmrs_scrm_id =
```

```
pusch_record.ulDmrsScramblingId dmrs_max_len = 1 dmrs_add_ln_pos = 1
num_dmrs_cdm_grps_no_data = pusch_record.numDmrsCdmGrpsNoData layers =
[pusch_record.nrOfLayers] scids = [pusch_record.SCID] dmrs_ports =
[pusch_record.dmrsPorts] slot = pusch_record.Slot tb_sizes =
[len(pusch_record.macPdu)] mod_orders = [pusch_record.qamModOrder]
code_rates = [pusch_record.targetCodeRate / 10240.] rvs = [0] ndis = [1] rntis =
[pusch_record.RNTI] data_scids = [pusch_record.dataScramblingId] ref_tb =
pusch_record.macPdu for llr_method in ["aerial", "llrnet", "logmap"]: if snr not in
tb_errors[llr_method].keys(): tb_errors[llr_method][snr] = 0 tb_count[llr_method]
[snr] = 0 receiver.set_llr_method(llr_method) tb = receiver.run( rx_slot=rx_slot,
num_ues=num_ues, slot=slot,
num_dmrs_cdm_grps_no_data=num_dmrs_cdm_grps_no_data,
dmrs_scrm_id=dmrs_scrm_id, start_prb=start_prb, num_prbs=num_prbs,
dmrs_syms=dmrs_syms, dmrs_max_len=dmrs_max_len,
dmrs_add_ln_pos=dmrs_add_ln_pos, start_sym=start_sym,
num_symbols=num_symbols, scids=scids, layers=layers, dmrs_ports=dmrs_ports,
rntis=rntis, data_scids=data_scids, code_rates=code_rates,
mod_orders=mod_orders, tb_sizes=[tb_sizes[0] * 8], rvs=rvs, ndis=ndis )
tb_count[llr_method][snr] += 1 tb_errors[llr_method][snr] += (not
np.array_equal(tb[:tb_sizes[0]], ref_tb[:tb_sizes[0]]))
```

[10]:

```
esno_dbs = tb_count["aerial"].keys() bler = dict(aerial=[], llrnet=[], logmap=[]) for
esno_db in esno_dbs: bler["aerial"].append(tb_errors["aerial"][esno_db] /
tb_count["aerial"][esno_db]) bler["llrnet"].append(tb_errors["llrnet"][esno_db] /
tb_count["llrnet"][esno_db]) bler["logmap"].append(tb_errors["logmap"][esno_db] /
tb_count["logmap"][esno_db])
```

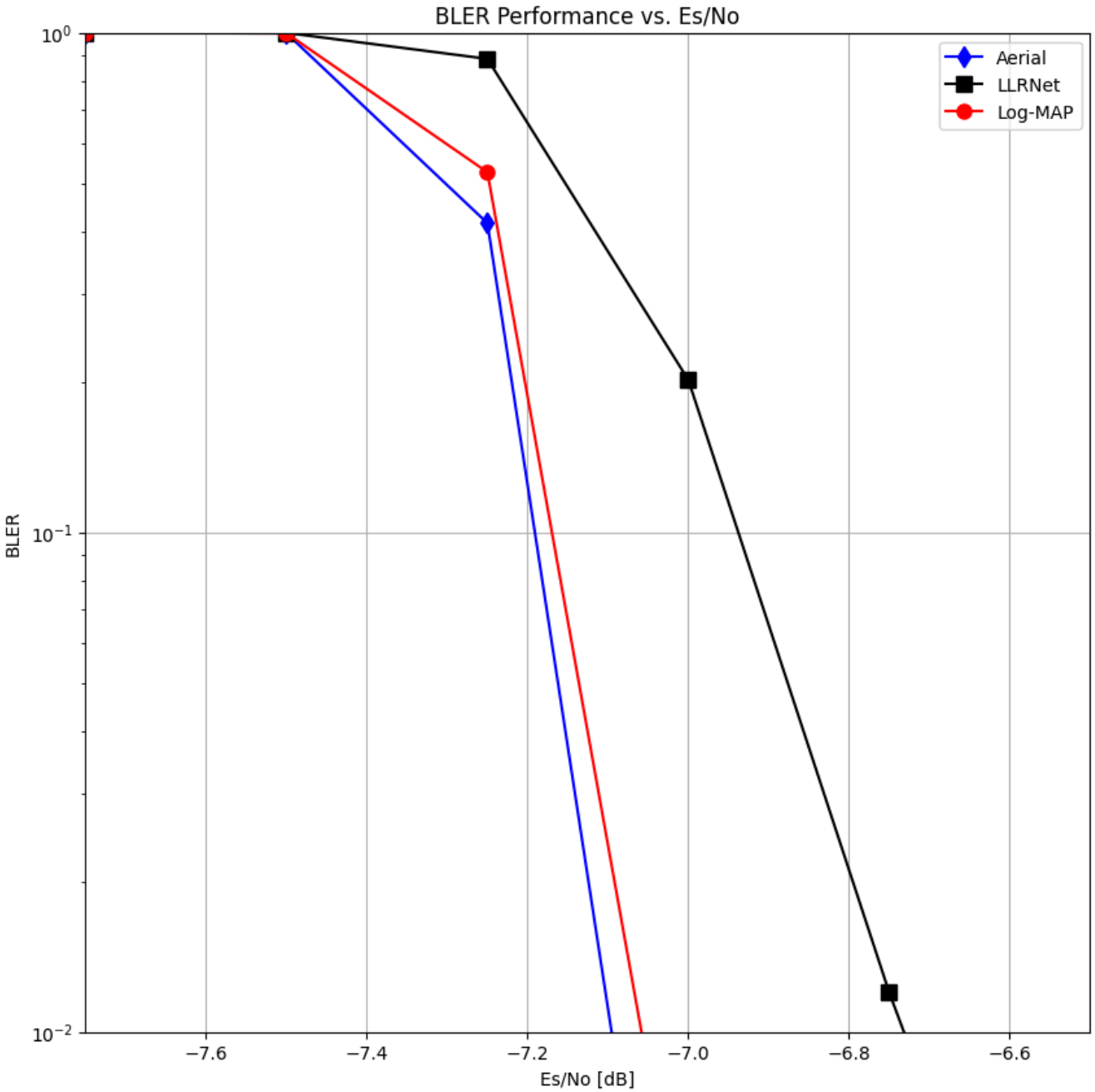
[11]:

```
esno_dbs = np.array(list(esno_dbs)) fig = plt.figure(figsize=(10, 10)) plt.yscale('log')
plt.ylim(0.01, 1) plt.xlim(np.min(esno_dbs), np.max(esno_dbs)) plt.title("BLER
Performance vs. Es/No") plt.ylabel("BLER") plt.xlabel("Es/No [dB]") plt.grid()
plt.plot(esno_dbs, bler["aerial"], marker="d", linestyle="-", color="blue",
markersize=8) plt.plot(esno_dbs, bler["llrnet"], marker="s", linestyle="-",
```

```
color="black", markersize=8) plt.plot(esno_dbs, bler["logmap"], marker="o",
linestyle="-", color="red", markersize=8) plt.legend(["Aerial", "LLRNet", "Log-MAP"])
```

[11]:

```
<matplotlib.legend.Legend at 0x7f3d681174c0>
```



## Export to TensorRT

Finally, the model gets exported to ONNX format to be consumed by the TensorRT inference engine.

[12]:

```
input_signature = [tf.TensorSpec([None, 2], tf.float16, name="input")] onnx_model, _ = tf2onnx.convert.from_keras(model, input_signature) onnx.save(onnx_model, "llrnet_QPSK_noscale.onnx") print("ONNX model created.")
```

```
ONNX model created.
```