Inatel

Trabalho final de AI/ML

Estimativa de canal e detecção de sinal em sistemas OFDM utilizando *Deep Learning*

Aluna: Mariana Baracat de Mello

Prof. Felipe Augusto Pereira de Figueiredo

23/06/2020

- Referência:

H. Ye, G. Y. Li and B. Juang, "**Power of Deep Learning for Channel Estimation and Signal Detection in OFDM Systems**," in *IEEE Wireless Communications Letters*, vol. 7, no. 1, pp. 114-117, Feb. 2018, doi: 10.1109/LWC.2017.2757490.

- Foram apresentados **resultados iniciais** em aprendizado profundo para estimativa de canal e detecção de sinal em sistemas de multiplexação por divisão de frequência ortogonal (OFDM).

Agenda

Inatel

1. Introdução
2. Estimação e detecção baseadas em *Deep Learning*
    1. Métodos de *Deep Learning*
    2. Arquitetura de sistema
    3. Modelo de treinamento
3. Resultados de simulação
4. Conclusão e sugestão para trabalhos futuros

Introdução

Inatel

- O esquema OFDM foi amplamente adotado em sistemas de banda larga sem fio para combater o **desvanecimento** em canais **seletivos em frequência**.
- A **informação sobre o estado do canal** de comunicação (*channel state information*, CSI) é vital para a detecção e decodificação coerentes nos sistemas OFDM.
- A CSI pode ser **estimada** por meio de **pilotos** antes da detecção.
- Com o CSI estimado, os símbolos transmitidos podem ser **recuperados** no receptor.
- A estimativa de canais em sistemas OFDM tem sido exaustivamente estudada.
- Métodos tradicionais de estimativa : mínimo quadrado (LS) e erro quadrático médio mínimo (MMSE).

Introdução

Inatel

- Este artigo apresenta uma abordagem de **aprendizado profundo** para **estimativa de canal** e **detecção de símbolo** em um sistema OFDM.
- É **treinado** um modelo de DNN para **prever** os dados transmitidos em diversas condições de canal.
- Em seguida, o modelo é usado na implantação *online* para recuperar os dados transmitidos.

Estimação e detecção baseadas em *Deep Learning* - Métodos de *Deep Learning*

Inatel

- Os DNNs são versões mais **profundas** de RNAs, aumentando o número de **camadas ocultas** para melhorar a capacidade de representação ou reconhecimento.
- Cada camada da rede consiste em vários neurônios, cada um com uma saída que é uma **função não-linear** (**função de ativação**) de uma **soma ponderada de neurônios** de sua camada anterior.
- A função não-linear pode ser uma **função Sigmoide**, ou uma **função Relu**.
- Portanto, a saída da rede **z** é uma **cascata** de transformação **não linear** dos dados de entrada **I.**

$$z = f(\boldsymbol{I}, \boldsymbol{\theta}) = f^{(L-1)}(f^{(L-2)}(\dots f^{(1)}(\boldsymbol{I})))$$

- Os parâmetros do modelo são os **pesos** para os neurônios, que precisam ser **otimizados** antes da implantação *online*.
- Os **pesos ideais** geralmente são aprendidos em um conjunto de treinamento, com resultados desejados conhecidos.

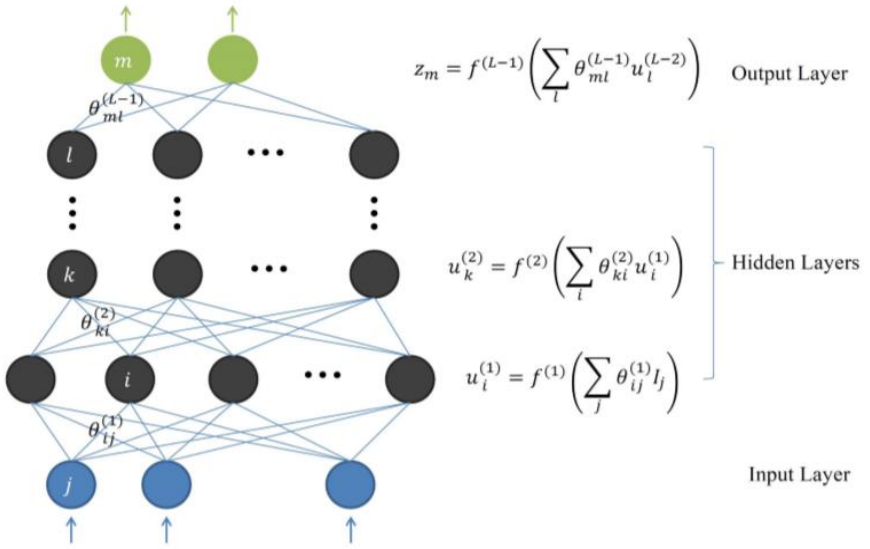


Fig. 1. An example of deep learning models.

Estimação e detecção baseadas em *Deep Learning* - Arquitetura do sistema

Inatel

- No lado do **transmissor**:
  - os **símbolos** transmitidos **inseridos** com os **pilotos** são primeiro convertidos em um **fluxo** de dados **paralelo**;
  - a **transformada discreta inversa de Fourier** (IDFT) é usada para converter o sinal do domínio da frequência no domínio do tempo.
  - Depois disso, um **prefixo cíclico** (CP) é inserido para **mitigar** a **interferência entre símbolos** (ISI).
- Considerou-se um canal com **múltiplos percursos** descrito por **variáveis aleatórias complexas**. De modo, que o sinal recebido é dado por

$$y(n) = x(n) \otimes h(n) + w(n)$$

- Após **remover** o CP e **executar** a DFT, o **sinal recebido** no domínio da **frequência** é dado por

$$Y(k) = X(k)H(k) + W(k)$$

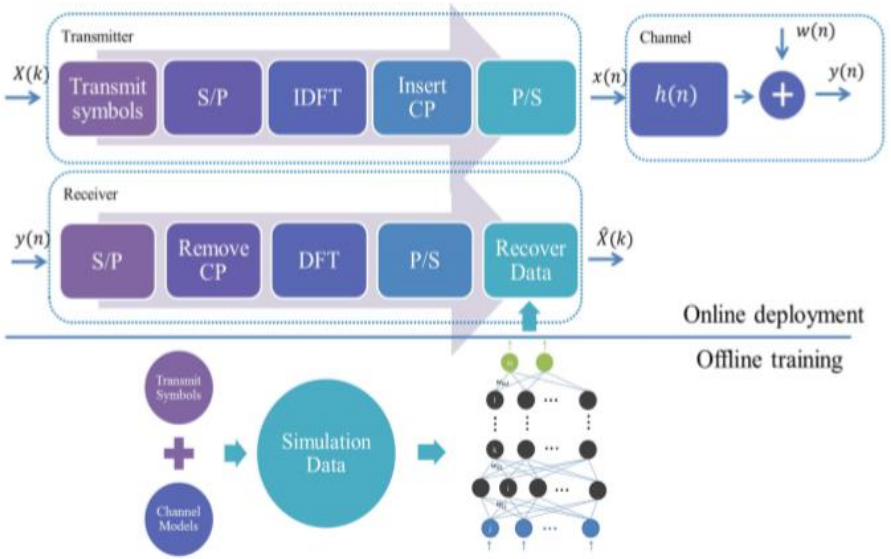


Fig. 2. System model.

Estimação e detecção baseadas em *Deep Learning* - Arquitetura do sistema

Inatel

- O **canal** pode ser tratado como **constante** sobre o **bloco** piloto e os blocos de dados, mas **muda** de um **quadro** para outro.
- O modelo DNN toma como **entrada** os dados **recebidos**, consistindo em um **bloco piloto** e um **bloco de dados** (equivalente a um ***frame***), e **recupera** os dados transmitidos de maneira **completa**.
- Para obter um modelo DNN **eficaz** para **conjuntamente** fazer a estimativa de canal e a detecção de símbolos, **dois estágios** são incluídos:
  - Estágio de **treinamento *offline***: o modelo é **treinado** com as **amostras** OFDM **recebidas** que são geradas com **diferentes sequências de informações** e sob **diversas condições de canal** com certas propriedades estatísticas.
  - Estágio de **implementação *online***: o modelo DNN gera a **saída** que **recupera** os **dados transmitidos sem estimar** explicitamente o **canal** sem fio.

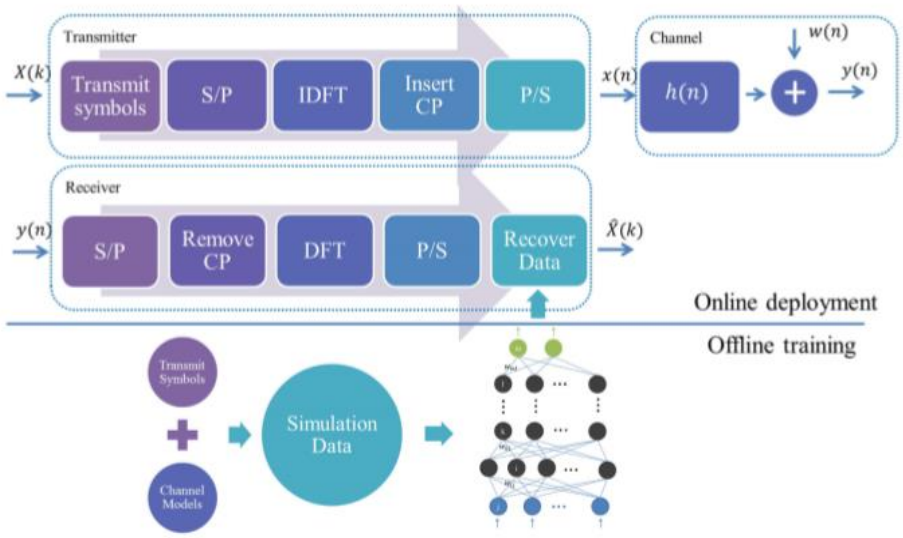


Fig. 2. System model.

Estimação e detecção baseadas em *Deep Learning* - Modelo de treinamento

Inatel

- Os modelos são treinados **visualizando** a modulação OFDM e os canais sem fio como **caixas pretas**.
- Em cada simulação, uma **sequência** de **dados aleatória** é gerada pela primeira vez com os **símbolos transmitidos** e o quadro OFDM **correspondente** é formado com **símbolos pilotos**. O **estado atual do canal aleatório** é simulado com base nos **modelos de canais reais**.
- O sinal OFDM recebido é obtido com base nos quadros OFDM submetidos à distorção do canal atual, **incluindo o ruído do canal**.
- O modelo é treinado para **minimizar** a **diferença** entre a saída da rede neural e os dados transmitidos.

$$L_2 = \frac{1}{N}\sum_{k}(\hat{X}(k) - X(k))^2$$

Estimação e detecção baseadas em *Deep Learning* - Modelo de treinamento

Inatel

- O modelo DNN que usamos consiste em **cinco camadas**, das quais **três** são **camadas ocultas**. O número de **neurônios** em cada camada é 256, 500, 250, 120, 16.
- O número de **entradas** corresponde ao número de partes **reais** e **imaginárias** de 2 **blocos** OFDM que contêm os **pilotos** e os **símbolos transmitidos**, respectivamente.
- A cada **16 bits** dos dados transmitidos são agrupados e **previstos** com base em um único modelo treinado de forma independente, que é concatenado para a saída final.
- A **função Relu** é usada como **função de ativação** na maioria das camadas, **exceto** na **última camada** em que a função **Sigmoide** é aplicada para **mapear a saída** para o **intervalo** [0,1].

Resultados de simulação

Inatel

- O **TensorFlow** oferece várias APIs.
- A API de **nível mais baixo** é o **TensorFlow Core**, que fornece o **controle** de programação **completo**.
- No entanto, APIs de **níveis superiores** são **mais fáceis** de **aprender** e **usar** do que o TensorFlow Core. Elas também tornam as tarefas **repetitivas mais fáceis** e **mais consistentes**.
- O ***tf.keras*** é uma API **alto-nível** para **construir** e **treinar** modelos no TensorFlow.

Resultados de simulação

Inatel

```
def ofdm_simulate(codeword, channelResponse, SNRdb):
    bits = np.random.binomial(n=1, p=0.5, size=(2*(K - P),))
    QAM = Modulation(bits)
    OFDM_data = np.zeros(K, dtype=complex)
    OFDM_data[pilotCarriers] = pilotValue
    OFDM_data[dataCarriers] = QAM
    OFDM_time = IDFT(OFDM_data)
    OFDM_withCP = addCP(OFDM_time)
    OFDM_TX = OFDM_withCP
    if Clipping_flag:
        OFDM_TX = Clipping(OFDM_TX,CR)
    OFDM_RX = channel(OFDM_TX, channelResponse, SNRdb)
    OFDM_RX_noCP = removeCP(OFDM_RX)
    OFDM_RX_noCP = DFT(OFDM_RX_noCP)

    # ----- target inputs ---
    symbol = np.zeros(K, dtype=complex)
    codeword_qam = Modulation(codeword)
    symbol[np.arange(K)] = codeword_qam
    OFDM_data_codeword = symbol
    OFDM_time_codeword = np.fft.ifft(OFDM_data_codeword)
    OFDM_withCP_cordword = addCP(OFDM_time_codeword)
    if Clipping_flag:
        OFDM_withCP_cordword = Clipping(OFDM_withCP_cordword,CR)
    OFDM_RX_codeword = channel(OFDM_withCP_cordword, channelResponse, SNRdb)
    OFDM_RX_noCP_codeword = removeCP(OFDM_RX_codeword)
    OFDM_RX_noCP_codeword = DFT(OFDM_RX_noCP_codeword)
    return np.concatenate(
        (np.concatenate(
            (np.real(OFDM_RX_noCP), np.imag(OFDM_RX_noCP))), np.concatenate(
            (np.real(OFDM_RX_noCP_codeword), np.imag(OFDM_RX_noCP_codeword))))), abs(channelResponse)
```

Resultados de simulação

Inatel

```
channel_train = np.load('channel_train.npy')
train_size = channel_train.shape[0]
channel_test = np.load('channel_test.npy')
test_size = channel_test.shape[0]


def training_gen(bs, SNRdb = 20):
    while True:
        index = np.random.choice(np.arange(train_size), size=bs)
        H_total = channel_train[index]
        input_samples = []
        input_labels = []
        for H in H_total:
            bits = np.random.binomial(n=1, p=0.5, size=(payloadBits_per_OFDM,))
            signal_output, para = ofdm_simulate(bits, H, SNRdb)
            input_labels.append(bits[0:16])
            input_samples.append(signal_output)
        yield (np.asarray(input_samples), np.asarray(input_labels))


def validation_gen(bs, SNRdb = 20):
    while True:
        index = np.random.choice(np.arange(test_size), size=bs)
        H_total = channel_test[index]
        input_samples = []
        input_labels = []
        for H in H_total:
            bits = np.random.binomial(n=1, p=0.5, size=(payloadBits_per_OFDM,))
            signal_output, para = ofdm_simulate(bits, H, SNRdb)
            input_labels.append(bits[0:16])
            input_samples.append(signal_output)
        yield (np.asarray(input_samples), np.asarray(input_labels))
```

Resultados de simulação

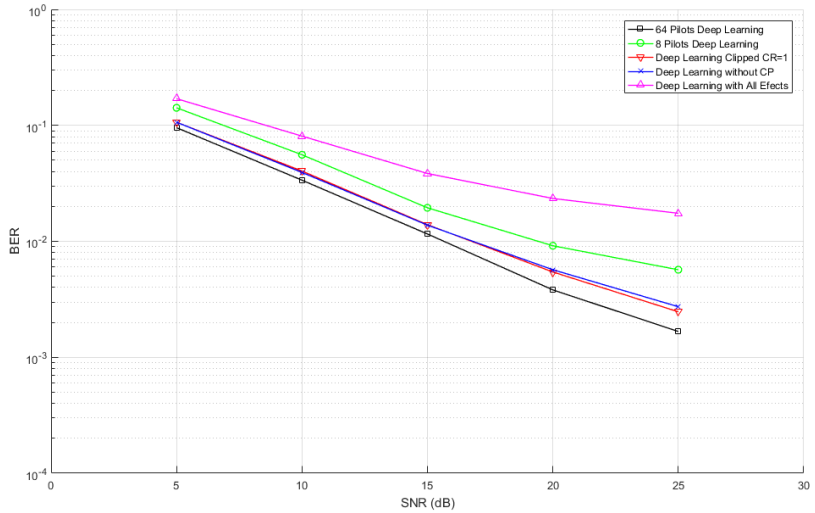
Inatel

```
def bit_err(y_true, y_pred):
    err = 1 - tf.reduce_mean(
        tf.reduce_mean(
            tf.to_float(
                tf.equal(
                    tf.sign(
                        y_pred - 0.5),
                    tf.cast(
                        tf.sign(
                            y_true - 0.5),
                        tf.float32))),
            1))
    return err

input_bits = Input(shape=(payloadBits_per_OFDM * 2,))
temp = BatchNormalization()(input_bits)
temp = Dense(n_hidden_1, activation='relu')(input_bits)
temp = BatchNormalization()(temp)
temp = Dense(n_hidden_2, activation='relu')(temp)
temp = BatchNormalization()(temp)
temp = Dense(n_hidden_3, activation='relu')(temp)
temp = BatchNormalization()(temp)
out_put = Dense(n_output, activation='sigmoid')(temp)
model = Model(input_bits, out_put)
model.compile(optimizer='RMSprop', loss='mse', metrics=[bit_err])
```

```
model.summary()
checkpoint = callbacks.ModelCheckpoint('./temp_trained_20_test.h5', monitor='val_bit_err',
                                       verbose=0, save_best_only=True, mode='min', save_weights_only=True)

model.fit_generator(
    training_gen(1000,20),
    steps_per_epoch=50,
    epochs=1000,
    validation_data=validation_gen(1000, 20),
    validation_steps=1,
    callbacks=[checkpoint],
    verbose=2)


model.load_weights('./temp_trained_20_test.h5')

BER = []
for SNR in range(5, 30, 5):
    y = model.evaluate(
        validation_gen(10000, SNR),
        steps=1
    )
    BER.append(y[1])
    print(y)
print(BER)
```

Resultados de simulação

Inatel



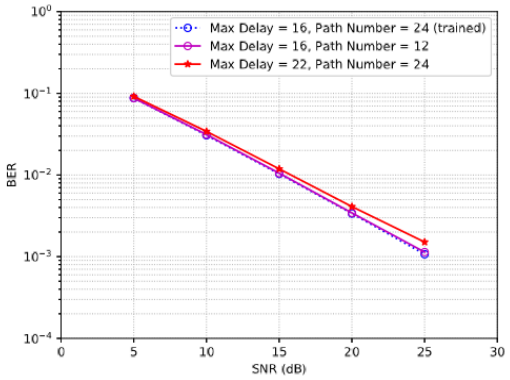


Fig. 7. BER curves with mismatch over training stage and deployment stage.

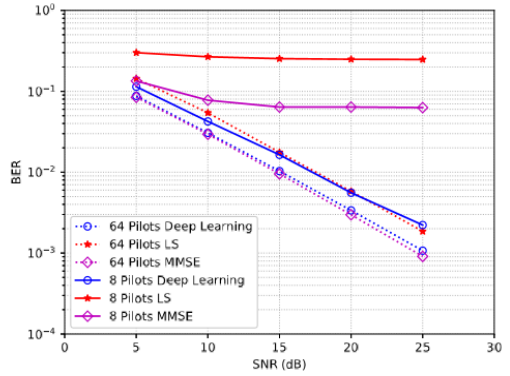


Fig. 3. BER curves of deep learning based approach and traditional methods.

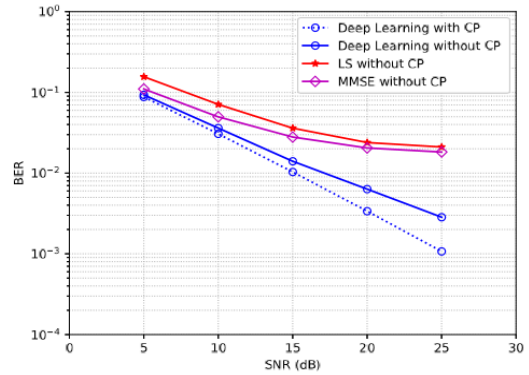


Fig. 4. BER curves without CP.

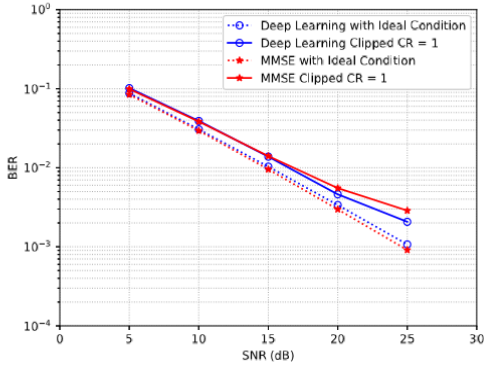


Fig. 5. BER curves with clipping noise

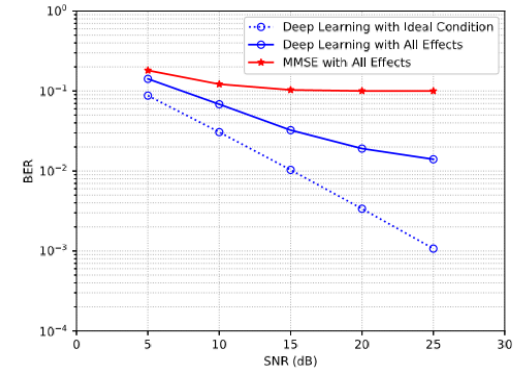


Fig. 6. BER curves when combining all adversities.

Conclusão e sugestão para trabalhos futuros

Inatel

- Os resultados da simulação mostram que o **método de aprendizado profundo** tem **vantagens** quando os canais sem fio sofrem **sérias distorções** e **interferências**, o que prova que as DNNs têm a **capacidade** de **lembrar e analisar** as **características complicadas** dos canais sem fio.
- Para aplicações no mundo **real**, é **importante** que o modelo DNN tenha uma boa capacidade de **generalização**, para que ainda possa **funcionar** efetivamente quando as **condições da implantação *online*** não estiverem **exatamente de acordo** com os **modelos de canal** usados na etapa de **treinamento**.
- **Aplicações em trabalhos futuros:** Análises em sistemas com **interferência intersímbólica intencional** (Sinalização *Faster than Nyquist*), onde **estimadores lineares não são capazes de estimar os símbolos** transmitidos sem causar **penalidades** na BER.

Obrigada !

Perguntas

Resultados de simulação

Inatel

```
from tensorflow.python.keras import *
from tensorflow.python.keras.layers import *
import tensorflow as tf
import os
import numpy as np

K = 64
CP = K//4
P = 64
CR = 1
allCarriers = np.arange(K)  # indices of all subcarriers ([0, 1, ... K-1])

if P < K:
    pilotCarriers = allCarriers[::K // P]  # Pilots is every (K/P)th carrier.
    dataCarriers = np.delete(allCarriers, pilotCarriers)

else:  # K = P
    pilotCarriers = allCarriers
    dataCarriers = []

mu = 2
payloadBits_per_OFDM = K * mu
SNRdb = 20
#H_folder_train = '../H_dataset/Train/'
#H_folder_test = '../H_dataset/Test/'
n_hidden_1 = 500
n_hidden_2 = 250  # 1st layer num features
n_hidden_3 = 120  # 2nd layer num features
n_output = 16  # every 16 bit are predicted by a model

Clipping_flag = False

def Clipping(x,CL):
    sigma = np.sqrt(np.mean(np.square(np.abs(x))))
    CL = CL*sigma
    x_clipped = x
    clipped_idx = abs(x_clipped) > CL
    x_clipped[clipped_idx] = np.divide(
                        (x_clipped[clipped_idx]*CL),abs(x_clipped[clipped_idx])
)
    return x_clipped

def Modulation(bits):
    bit_r = bits.reshape((int(len(bits) / mu), mu))
    # This is just for QAM modulation
    return (2 * bit_r[:, 0] - 1) + 1j * (2 * bit_r[:, 1] - 1)


def OFDM_symbol(Data, pilot_flag):
    symbol = np.zeros(K, dtype=complex)  # the overall K subcarriers
    #symbol = np.zeros(K)
    symbol[pilotCarriers] = pilotValue  # allocate the pilot subcarriers
    symbol[dataCarriers] = Data  # allocate the pilot subcarriers
    return symbol


def IDFT(OFDM_data):
    return np.fft.ifft(OFDM_data)


def addCP(OFDM_time):
    cp = OFDM_time[-CP:]            # take the last CP samples ...
    return np.hstack([cp, OFDM_time])  # ... and add them to the beginning

def channel(signal, channelResponse, SNRdb):
    convolved = np.convolve(signal, channelResponse)
    signal_power = np.mean(abs(convolved**2))
    sigma2 = signal_power * 10**(-SNRdb / 10)
    noise = np.sqrt(sigma2 / 2) * (np.random.randn(*
                                   convolved.shape)
                                   + 1j * np.random.randn(*convolved.shape))
    return convolved + noise
```

Resultados de simulação

Inatel

```
def removeCP(signal):
    return signal[CP:(CP + K)]


def DFT(OFDM_RX):
    return np.fft.fft(OFDM_RX)


def ofdm_simulate(codeword, channelResponse, SNRdb):
    bits = np.random.binomial(n=1, p=0.5, size=(2*(K - P),))
    QAM = Modulation(bits)
    OFDM_data = np.zeros(K, dtype=complex)
    OFDM_data[pilotCarriers] = pilotValue
    OFDM_data[dataCarriers] = QAM
    OFDM_time = IDFT(OFDM_data)
    OFDM_withCP = addCP(OFDM_time)
    OFDM_TX = OFDM_withCP
    if Clipping_flag:
        OFDM_TX = Clipping(OFDM_TX,CR)
    OFDM_RX = channel(OFDM_TX, channelResponse, SNRdb)
    OFDM_RX_noCP = removeCP(OFDM_RX)
    OFDM_RX_noCP = DFT(OFDM_RX_noCP)

    # ----- target inputs ---
    symbol = np.zeros(K, dtype=complex)
    codeword_qam = Modulation(codeword)
    symbol[np.arange(K)] = codeword_qam
    OFDM_data_codeword = symbol
    OFDM_time_codeword = np.fft.ifft(OFDM_data_codeword)
    OFDM_withCP_cordword = addCP(OFDM_time_codeword)
    if Clipping_flag:
        OFDM_withCP_cordword = Clipping(OFDM_withCP_cordword,CR)
    OFDM_RX_codeword = channel(OFDM_withCP_cordword,
                               channelResponse, SNRdb)
    OFDM_RX_noCP_codeword = removeCP(OFDM_RX_codeword)
    OFDM_RX_noCP_codeword = DFT(OFDM_RX_noCP_codeword)
    return np.concatenate(
        (np.concatenate(
            (np.real(OFDM_RX_noCP), np.imag(OFDM_RX_noCP))),
                    np.concatenate(
            (np.real(OFDM_RX_noCP_codeword),
                    np.imag(OFDM_RX_noCP_codeword)))),
                               abs(channelResponse))


Pilot_file_name = 'Pilot_' + str(P)
if os.path.isfile(Pilot_file_name):
    print('Load Training Pilots txt')
    # load file
    bits = np.loadtxt(Pilot_file_name, delimiter=',')
else:
    # write file
    bits = np.random.binomial(n=1, p=0.5, size=(K * mu, ))
    np.savetxt(Pilot_file_name, bits, delimiter=',')

pilotValue = Modulation(bits)
```

Resultados de simulação

Inatel

```
channel_train = np.load('channel_train.npy')
train_size = channel_train.shape[0]
channel_test = np.load('channel_test.npy')
test_size = channel_test.shape[0]


def training_gen(bs, SNRdb = 20):
    while True:
        index = np.random.choice(np.arange(train_size), size=bs)
        H_total = channel_train[index]
        input_samples = []
        input_labels = []
        for H in H_total:
            bits = np.random.binomial(n=1, p=0.5, size=(payloadBits_per_OFDM,))
            signal_output, para = ofdm_simulate(bits, H, SNRdb)
            input_labels.append(bits[0:16])
            input_samples.append(signal_output)
        yield (np.asarray(input_samples), np.asarray(input_labels))


def validation_gen(bs, SNRdb = 20):
    while True:
        index = np.random.choice(np.arange(test_size), size=bs)
        H_total = channel_test[index]
        input_samples = []
        input_labels = []
        for H in H_total:
            bits = np.random.binomial(n=1, p=0.5, size=(payloadBits_per_OFDM,))
            signal_output, para = ofdm_simulate(bits, H, SNRdb)
            input_labels.append(bits[0:16])
            input_samples.append(signal_output)
        yield (np.asarray(input_samples), np.asarray(input_labels))
```

Resultados de simulação

Inatel

```
def bit_err(y_true, y_pred):
    err = 1 - tf.reduce_mean(
        tf.reduce_mean(
            tf.to_float(
                tf.equal(
                    tf.sign(
                        y_pred - 0.5),
                    tf.cast(
                        tf.sign(
                            y_true - 0.5),
                        tf.float32))),
            1))
    return err

input_bits = Input(shape=(payloadBits_per_OFDM * 2,))
temp = BatchNormalization()(input_bits)
temp = Dense(n_hidden_1, activation='relu')(input_bits)
temp = BatchNormalization()(temp)
temp = Dense(n_hidden_2, activation='relu')(temp)
temp = BatchNormalization()(temp)
temp = Dense(n_hidden_3, activation='relu')(temp)
temp = BatchNormalization()(temp)
out_put = Dense(n_output, activation='sigmoid')(temp)
model = Model(input_bits, out_put)
model.compile(optimizer='RMSprop', loss='mse', metrics=[bit_err])
```

```
model.summary()
checkpoint = callbacks.ModelCheckpoint('./temp_trained_20_test.h5', monitor='val_bit_err',
                                       verbose=0, save_best_only=True, mode='min', save_weights_only=True)

model.fit_generator(
    training_gen(1000,20),
    steps_per_epoch=50,
    epochs=1000,
    validation_data=validation_gen(1000, 20),
    validation_steps=1,
    callbacks=[checkpoint],
    verbose=2)


model.load_weights('./temp_trained_20_test.h5')

BER = []
for SNR in range(5, 30, 5):
    y = model.evaluate(
        validation_gen(10000, SNR),
        steps=1
    )
    BER.append(y[1])
    print(y)
print(BER)
```