FACULDADE DE CIÊNCIAS E TECNOLOGIA
UNIVERSIDADE D COIMBRA

Técnicas Avançadas de Análise de Dados
Prof. D.º José Ricardo Gonçalo
2021/2022

TileCal Degradation

Machine Learning for detector design, modelling the TileCal degradation history as a use case

Orientadora: D.ª Rute Pedro
Hugo Filipe Costa, 2016224841
Seomara Félix, 2017253011

Table of Contents

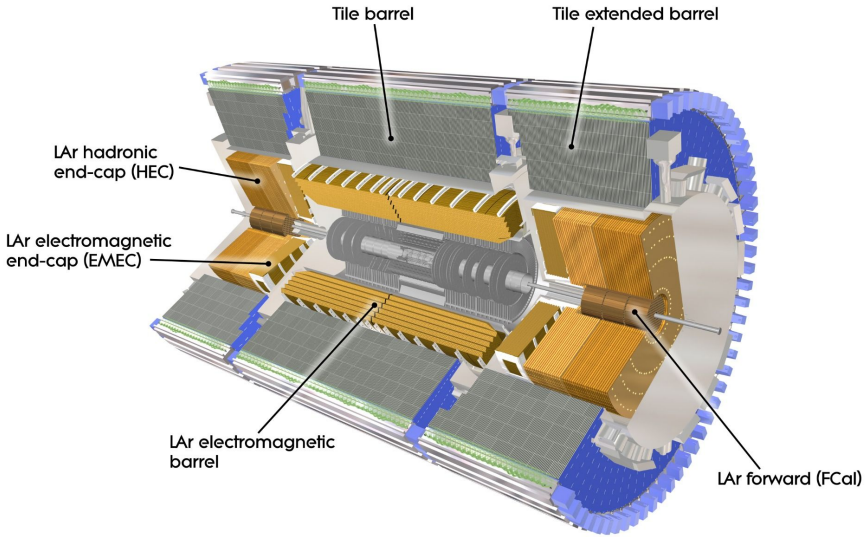
01

Introduction

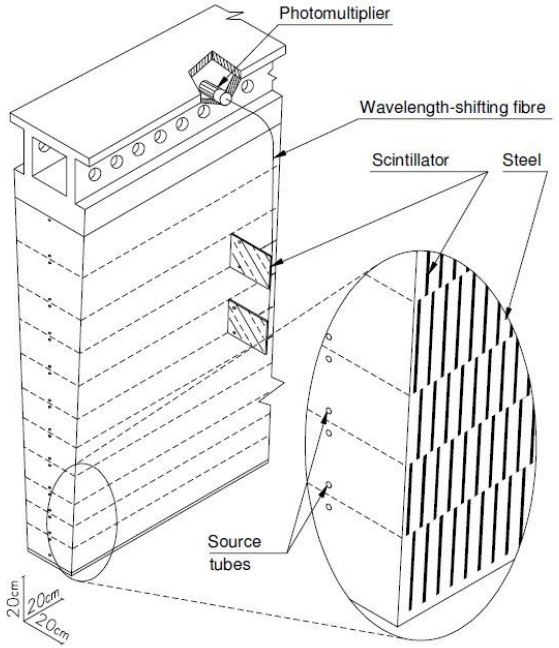
TileCal working principle, geometry, and degradation.

Introduction

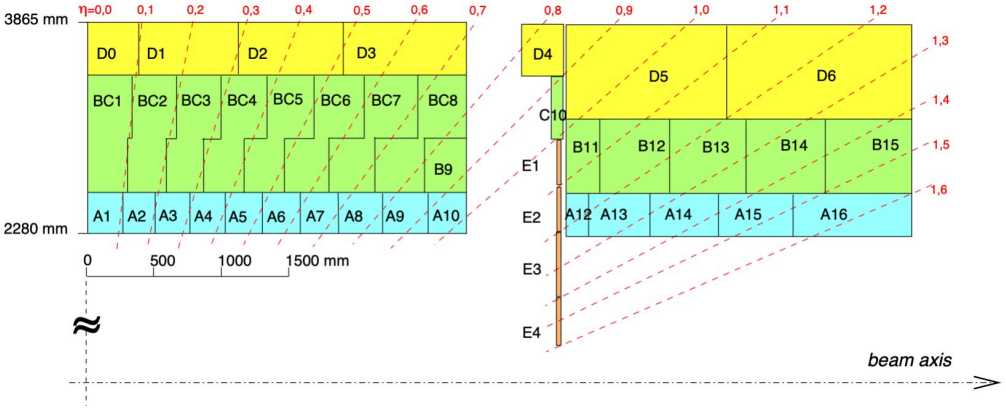
The hadronic TileCal Calorimeter is one of the sub-detectors of ATLAS and contributes to the measurement of jets, missing energy and in the trigger decision [1, 2].



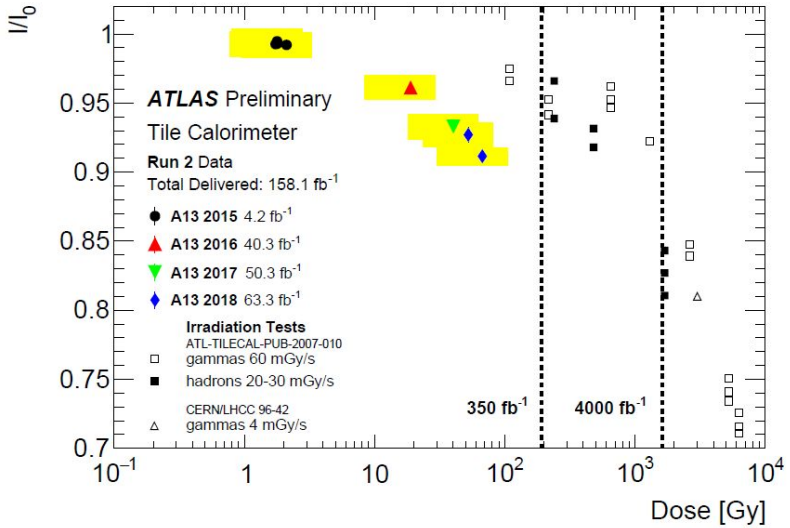
Geometry



The detector consists of 64 modules (left image). The detector's readout unit is one cell: A, B, D (right image).



“The objective is to model the degradation of the TileCal scintillators and optical fibres with Machine Learning regression to investigate the dependencies on the specific detector design parameters and run conditions.”



02

Model Training

Tuning the hyperparameters and training the model

Selecting and clearing the data

```
np.random.seed(0)
tf.random.set_seed(0)

data = pd.read_csv("https://www.lip.pt/~rute/TileAgeing/LightLoss_2015201620172018.csv",delimiter=" ")

input_features = ['nlayers', 'ntiles', 'nshortfibres',
        'nlongfibres', 'lminfibre', 'lmaxfibre',
        'tileheight', 'lmintile', 'lmaxtile',
        'doserate', 'doserateRMS', 'date',
        'intluminosity', 'dose', 'doseRMS']

train_cells=[]
test_cells =[]
counter=0

for cell in data['cell'].unique():

    counter+=1

    if cell in ['D5','D6']: continue
    #volta ao inicio do loop
    #problemas nos dados delas

    even = (int(cell[-1])+1)%3==0
    #print(f'{cell} is even? {even}')
    if even:
      test_cells.append(cell)
      #print(cell[-1])
    else: train_cells.append(cell)

data_clean = data.query('lightyield < 1.01 & lightyield > 0.6') # some clean-up of residual defectuous data that we know about (malfunctioning channels etc...)
#estes são os valores que mantemos,abaixo de 1.01 e acima de 0.6 são removidos

print(f'Clean data samples {len(data_clean)}/{len(data)}')

train_data = data_clean.query('cell in @train_cells')
test_data  = data_clean.query('cell in @test_cells')
```

Selecting and clearing the data

```
np.random.seed(0)
tf.random.set_seed(0)

data = pd.read_csv("https://www.lip.pt/~rute/TileAgeing/LightLoss_2015201620172018.csv",delimiter=" ")

input_features = ['nlayers', 'ntiles', 'nshortfibres',
       'nlongfibres', 'lminfibre', 'lmaxfibre',
       'tileheight', 'lmintile', 'lmaxtile',
       'doserate', 'doserateRMS', 'date',
       'intluminosity', 'dose', 'doseRMS']

train_cells=[]
test_cells =[]
counter=0

for cell in data['cell'].unique():

    counter+=1

    if cell in ['D5','D6']: continue
    #volta ao inicio do loop
    #problemas nos dados delas

    even = (int(cell[-1])+1)%3==0
    #print(f'{cell} is even? {even}')
    if even:
      test_cells.append(cell)
      #print(cell[-1])
    else: train_cells.append(cell)

data_clean = data.query('lightyield < 1.01 & lightyield > 0.6') # some clean-up of residual defectuous data that we know about (malfunctioning channels etc...)
#estes são os valores que mantemos,abaixo de 1.01 e acima de 0.6 são removidos

print(f'Clean data samples {len(data_clean)}/{len(data)}')

train_data = data_clean.query('cell in @train_cells')
test_data  = data_clean.query('cell in @test_cells')
```

Selecting and clearing the data

```
np.random.seed(0)
tf.random.set_seed(0)

data = pd.read_csv("https://www.lip.pt/~rute/TileAgeing/LightLoss_2015201620172018.csv",delimiter=" ")

input_features = ['nlayers', 'ntiles', 'nshortfibres',
        'nlongfibres', 'lminfibre', 'lmaxfibre',
        'tileheight', 'lmintile', 'lmaxtile',
        'doserate', 'doserateRMS', 'date',
        'intluminosity', 'dose', 'doseRMS']

train_cells=[]
test_cells =[]
counter=0

for cell in data['cell'].unique():

    counter+=1

    if cell in ['D5','D6']: continue
    #volta ao inicio do loop
    #problemas nos dados delas

    even = (int(cell[-1])+1)%3==0
    #print(f'{cell} is even? {even}')
    if even:
      test_cells.append(cell)
      #print(cell[-1])
    else: train_cells.append(cell)
```

```
train cells  ['A1', 'A2', 'A4', 'A5', 'A7', 'A8', 'A12', 'A14', 'A15', 'BC1', 'BC2', 'BC4', 'BC5', 'BC7', 'BC8', 'B11', 'B12', 'B14', 'B15', 'D1', 'D2', 'D4']
test cells  ['A3', 'A6', 'A9', 'A10', 'A13', 'A16', 'BC3', 'BC6', 'B9', 'C10', 'B13', 'D0', 'D3']
We have 35 usable cells train cells are 62.857142857142854 %, test cells are 37.142857142857146 %
```

```
train_data = data_clean.query('cell in @train_cells')
test_data  = data_clean.query('cell in @test_cells')
```

Scale the data

```
from sklearn.preprocessing import StandardScaler

# Standardizar
scaler=StandardScaler()
# Os outputs (lightyield) não devem ser normalizados; Nos dados de treino usar fit_transform e nos de test transform
scaler.fit(train_data[input_features].values)
train_data_std_f=scaler.transform(train_data[input_features].values)
test_data_std_f=scaler.transform(test_data[input_features].values)
```

$$z = \frac{x - u}{\sigma}$$

The activation function, namely the sigmoid, react better if the input data is normalized between -1 and 1.

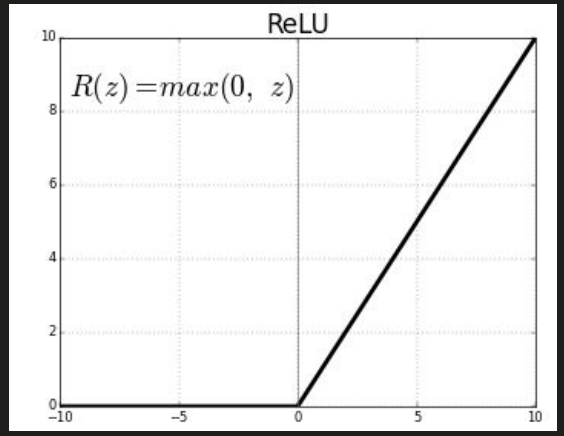
Normalization ensures that the magnitude of the values that a feature assumes are approximately the same, and consequently the weights of the inputs.

DNN creation

```
i=20
NN_model_array = [];
while(i<111):
    NN_model = Sequential();

    # The Input Layer
    NN_model.add(Dense(i, kernel_initializer='normal',input_dim = train_data[input_features].shape[1], activation='relu'))


    # The Hidden Layers
    NN_model.add(Dense(i, kernel_initializer='normal',activation='relu'))
    NN_model.add(Dense(i, kernel_initializer='normal',activation='relu'))
    NN_model.add(Dense(i, kernel_initializer='normal',activation='relu'))


    #NN_model.add(LayerNormalization())

    # The Output Layer
    NN_model.add(Dense(1, kernel_initializer='normal',activation='sigmoid'))


    # Compile the network

    NN_model.compile(optimizer=tf.keras.optimizers.Adam(learning_rate=0.0001), loss='mae', metrics=['mae'])
    NN_model.build(train_data[input_features].shape)

    #NN_model.summary()


    NN_model_array.append(NN_model)
    i+=10
```

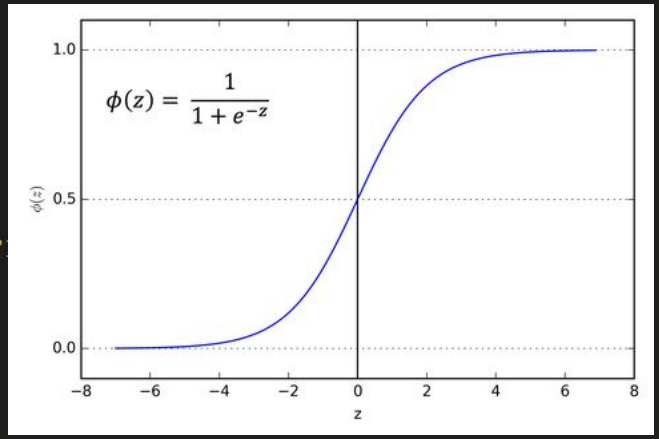
DNN creation

```
i=20
NN_model_array = [];
while(i<111):
    NN_model = Sequential();

    # The Input Layer
    NN_model.add(Dense(i, kernel_initializer='normal',input_dim = train_data[input_features].shape[1], activation='relu'))


    # The Hidden Layers
    NN_model.add(Dense(i, kernel_initializer='normal',activation='relu'))
    NN_model.add(Dense(i, kernel_initializer='normal',activation='relu'))
    NN_model.add(Dense(i, kernel_initializer='normal',activation='relu'))


    #NN_model.add(LayerNormalization())

    # The Output Layer
    NN_model.add(Dense(1, kernel_initializer='normal',activation='sigmoid'))


    # Compile the network

    NN_model.compile(optimizer=tf.keras.optimizers.Adam(learning_rate=0.0001), loss='mae', metrics=['mae'])
    NN_model.build(train_data[input_features].shape)

    #NN_model.summary()


    NN_model_array.append(NN_model)
    i+=10
```



DNN creation

```
i=20
NN_model_array = [];
while(i<111):
    NN_model = Sequential();

    # The Input Layer
    NN_model.add(Dense(i, kernel_initializer='normal',input_dim = train_data[input_features].shape[1], activation='relu'))


    # The Hidden Layers
    NN_model.add(Dense(i, kernel_initializer='normal',activation='relu'))
    NN_model.add(Dense(i, kernel_initializer='normal',activation='relu'))
    NN_model.add(Dense(i, kernel_initializer='normal',activation='relu'))


    #NN_model.add(LayerNormalization())

    # The Output Layer
    NN_model.add(Dense(1, kernel_initializer='normal',activation='sigmoid'))


    # Compile the network

    NN_model.compile(optimizer=tf.keras.optimizers.Adam(learning_rate=0.0001), loss='mae', metrics=['mae'
    NN_model.build(train_data[input_features].shape)

    #NN_model.summary()


    NN_model_array.append(NN_model)
    i+=10
```



DNN creation

```
i=20
NN_model_array = [];
while(i<111):
    NN_model = Sequential();

    # The Input Layer
    NN_model.add(Dense(i, kernel_initializer='normal',input_dim = train_data[input_features].shape[1], activation='relu'))


    # The Hidden Layers
    NN_model.add(Dense(i, kernel_initializer='normal',activation='relu'))
    NN_model.add(Dense(i, kernel_initializer='normal',activation='relu'))
    NN_model.add(Dense(i, kernel_initializer='normal',activation='relu'))


    #NN_model.add(LayerNormalization())

    # The Output Layer
    NN_model.add(Dense(1, kernel_initializer='normal',activation='sigmoid'))


    # Compile the network

    NN_model.compile(optimizer=tf.keras.optimizers.Adam(learning_rate=0.0001), loss='mae', metrics=['mae'])
    NN_model.build(train_data[input_features].shape)

    #NN_model.summary()


    NN_model_array.append(NN_model)
    i+=10
```

$$\mathrm{MAE} = \frac{1}{n}\sum_{i=1}^{n} |x_i - x|$$

Fitting the model

```
NN_model=NN_model_array[5]
fit_model = NN_model_array[5].fit(train_data_std_f, train_data['lightyield'].values, epochs=6, batch_size=32, verbose=1,validation_split = 0.5, shuffle=True, callbacks=EarlyStopping(patience=2))
y_pred = NN_model_array[5].predict(test_data_std_f,verbose=0)
```

```
train_data_std_f, train_data['lightyield'].values,
```

```
epochs=6,
```

```
batch_size=32,
```

```
validation_split = 0.5,
```

```
callbacks=EarlyStopping(patience=2)
```

Fitting the model

```
NN_model=NN_model_array[5]
fit_model = NN_model_array[5].fit(train_data_std_f, train_data['lightyield'].values, epochs=6, batch_size=32, verbose=1,validation_split = 0.5, shuffle=True, callbacks=EarlyStopping(patience=2))
y_pred = NN_model_array[5].predict(test_data_std_f,verbose=0)
```

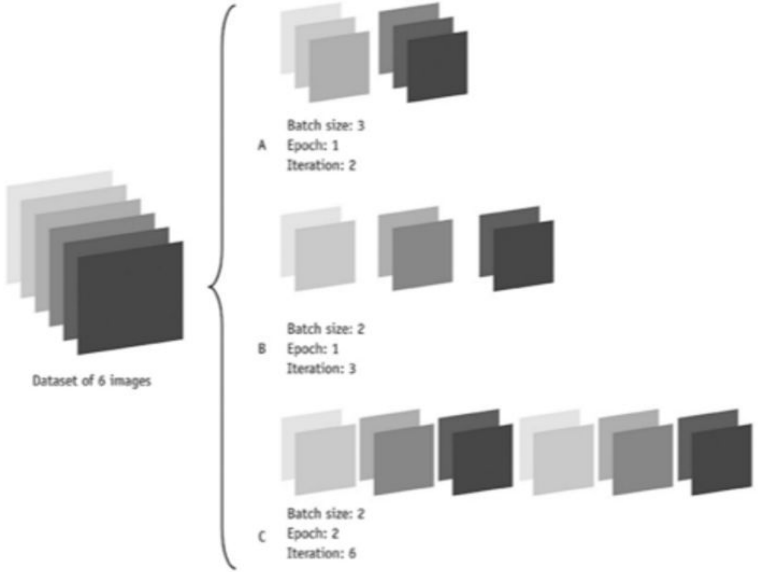
```
train_data_std_f, train_data['lightyield'].values,
```

```
epochs=6,
```

```
batch_size=32,
```

```
validation_split = 0.5,
```

```
callbacks=EarlyStopping(patience=2)
```



Fitting the model

```
NN_model=NN_model_array[5]
fit_model = NN_model_array[5].fit(train_data_std_f, train_data['lightyield'].values, epochs=6, batch_size=32, verbose=1,validation_split = 0.5, shuffle=True, callbacks=EarlyStopping(patience=2))
y_pred = NN_model_array[5].predict(test_data_std_f,verbose=0)
```

```
train_data_std_f, train_data['lightyield'].values,
```

```
epochs=6,
```

```
batch_size=32,
```

```
validation_split = 0.5,
```

```
callbacks=EarlyStopping(patience=2)
```

Choosing the model

```
i=1
NN_model_array = [];
while(i<110):
    NN_model = Sequential();

    # The Input Layer
    NN_model.add(Dense(50, kernel_initializer='normal',input_dim = train_data[input_features].shape[1],
                       activation='relu'))


    # The Hidden Layers
    NN_model.add(Dense(i, kernel_initializer='normal',activation='relu'))
    NN_model.add(Dense(i, kernel_initializer='normal',activation='relu'))


    #NN_model.add(LayerNormalization())

    # The Output Layer
    NN_model.add(Dense(1, kernel_initializer='normal',activation='sigmoid'))


    # Compile the network

    NN_model.compile(optimizer=tf.keras.optimizers.Adam(learning_rate=0.0001),
                     loss='mae', metrics=['mae'])
    NN_model.build(train_data[input_features].shape)

    #NN_model.summary()


    NN_model_array.append(NN_model)
    i+=10
```

Tests done:

- 0 to 5 layers
- 1 to 101 neurons
  - First layer fixed on 50 neurons and others changing;
  - All layers changing simultaneously.

```
i=0
desvio_padrão_esperado=test_data['lightyield'].values.flatten().std()
duration_array =[];
val_loss_array =[];
loss_array =[];
std_array =[];

while i<len(NN_model_array):
    np.random.seed(0)
    tf.random.set_seed(0)
    start = time.time()
    fit_model = NN_model_array[i].fit(train_data_std_f, train_data['lightyield'].values,
    epochs=1000, batch_size=32, verbose=0,validation_split = 0.5, shuffle=True, callbacks=EarlyStopping(patience=2))
    end = time.time()

    y_pred = NN_model_array[i].predict(test_data_std_f,verbose=0)
    y_pred_flat=y_pred.flatten()

    # duration_array.append(end-start)
    # loss_array.append(fit_model.history['loss'])
    # val_loss_array.append(fit_model.history['val_loss'])
    std_array.append(abs(desvio_padrão_esperado-y_pred_flat.std()))

    i+=1
```

Choosing the model:

```
O valor foi atingido para  70 neuronios com valor mínimo de desvio de 0.01004837826692663
70
O valor foi atingido para  60 neuronios com valor mínimo de desvio de 0.0058934167107304175
60
O valor foi atingido para  30 neuronios com valor mínimo de desvio de 0.0033344784831365187
30
O valor foi atingido para  30 neuronios com valor mínimo de desvio de 4.847067541279004e-05
30
O valor foi atingido para  70 neuronios com valor mínimo de desvio de 0.0007945383047422011
70
[70, 60, 30, 30, 70]


O valor mínimo foi encontrado para 4 camadas com 30 neurónios
```

```
0
O mínimo for encontrado para um número de neurónios= 20 com valor de perda= 0.02195674739778042
O valor foi atingido ao final de 12 epoch de um total de  14
4
O mínimo for encontrado para um número de neurónios= 60 com valor de perda= 0.014074659906327724
O valor foi atingido ao final de 8 epoch de um total de  10
5
O mínimo for encontrado para um número de neurónios= 70 com valor de perda= 0.012914612889289856
O valor foi atingido ao final de 6 epoch de um total de  8
5
O mínimo for encontrado para um número de neurónios= 70 com valor de perda= 0.009086228907108307
O valor foi atingido ao final de 6 epoch de um total de  8
5
O mínimo for encontrado para um número de neurónios= 70 com valor de perda= 0.00993399042636156
O valor foi atingido ao final de 6 epoch de um total de  8


O valor mínimo foi encontrado para 4 camadas com 70 neurónios
```

Tuning the Hyperparameters

The **hyperparameters** that correspond to a better performance of the model are:

- Four hidden layers;
- Seventy nodes for each layer.

Learning rate

The learning rate is the factor that controls the learning speed. The optimized value is 0,0001.

Activation Functions

For the input and hidden layers we use the ReLu function, and for the output layer we use the sigmoid function.

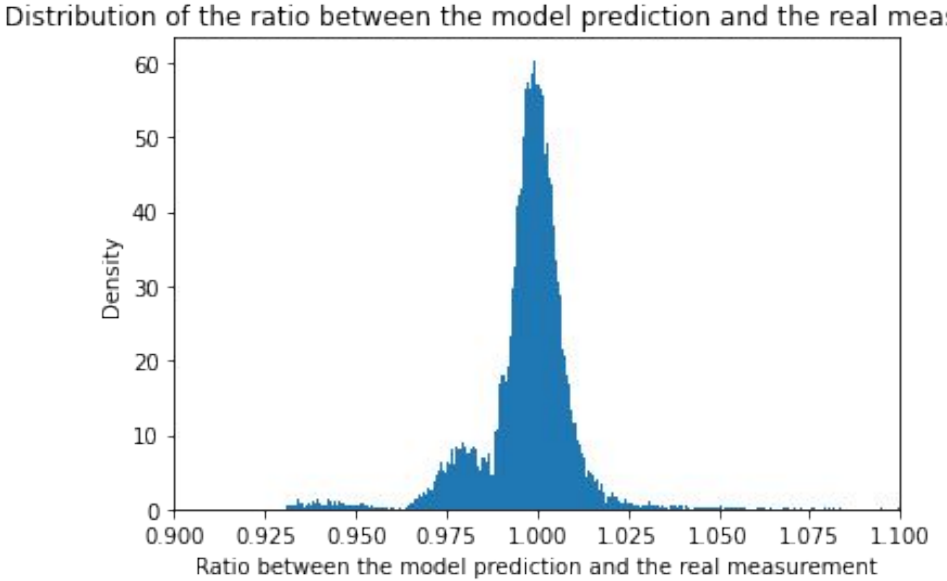
Normalization

StandardScale Function.

Metric

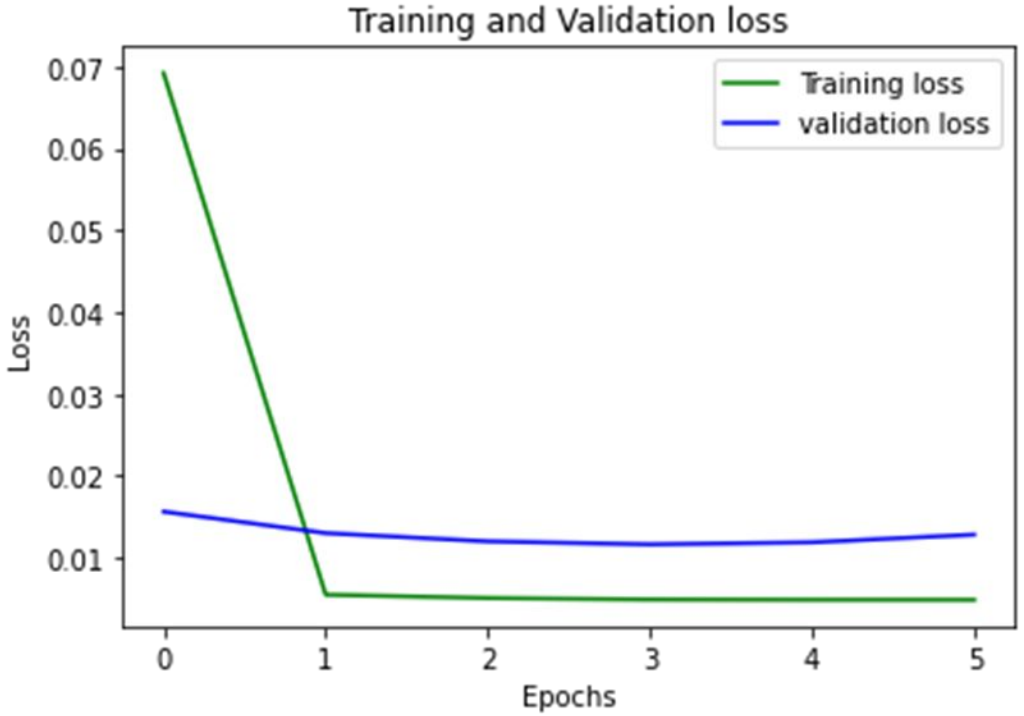
Mean Absolute Error.

Model Prediction and Real Measurement



	Mean	Standard Deviation
Predicted values of lightyiedl	0.991219	0.017363
Measured values of lightyield	0.995287	0.012725

Loss Function

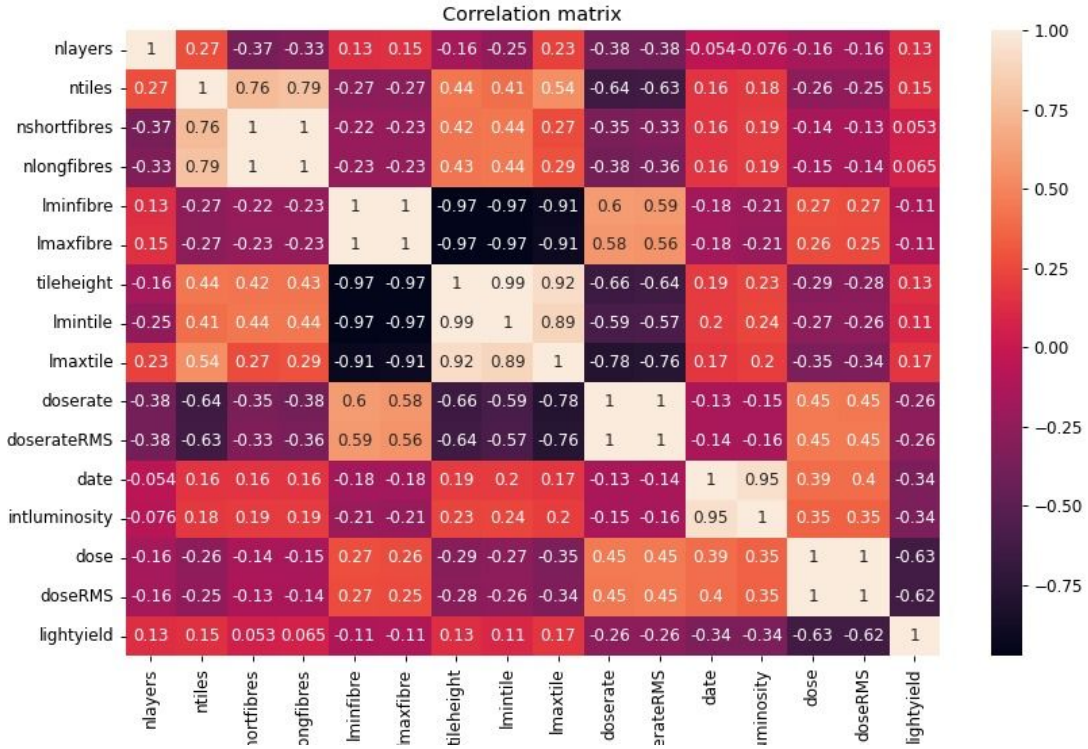


03

Model Dependence

Investigation on the geometry parameters model dependence

Correlation Matrix



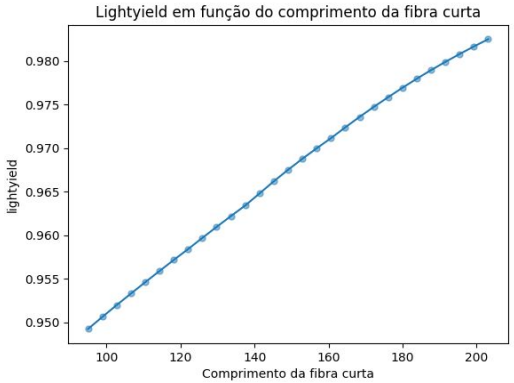
```python
def correlation_simul(min_value,max_value,nr_intervals,tested_colun,NN_model,mean_array,std_array,index,data_matrix):
    nr_short_fiber_tested=nr_intervals

    min_value=data_matrix[:,tested_colun].min()
    max_value=data_matrix[:,tested_colun].max()


    interval_of_values=abs(min_value)+abs(max_value)
    increment_short=(interval_of_values/nr_short_fiber_tested)


    #pego numa linha dos dados de teste, neste caso na linha
    start_array=data_matrix[index]

    #matriz dos dados que vamos usar para fazer a previsão
    start_matrix=np.zeros((nr_short_fiber_tested,15))
    #para ter um array das sequencias
    short_fiber_array=np.zeros(nr_short_fiber_tested)
    real_short_fiber_array=np.zeros(nr_short_fiber_tested)

    i=0
    #meto as linhas da matriz todas iguais ao start array
    while(i<len(start_matrix)):
        start_matrix[i]=start_array
        i+=1



    i=0
    nr_short_fiber=min_value
    while(i<len(short_fiber_array)):
        short_fiber_array[i]=nr_short_fiber
        real_short_fiber_array[i]=nr_short_fiber*std_array[tested_colun]+mean_array[tested_colun]
        nr_short_fiber+=increment_short
        i+=1


    nr_short_fiber=min_value
    nr_short_fiber_i=0
    while(nr_short_fiber_i<nr_short_fiber_tested):
        start_matrix[nr_short_fiber_i,tested_colun]=nr_short_fiber
        nr_short_fiber+=increment_short
        nr_short_fiber_i+=1

    y_pred_fiber = NN_model.predict(start_matrix)
    y_pred_fiber_flat=y_pred_fiber.flatten()



    return(short_fiber_array,y_pred_fiber_flat,real_short_fiber_array)
```

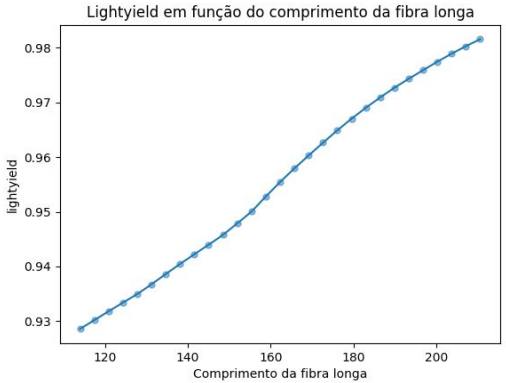
```python
def correlation_simul(min_value,max_value,nr_intervals,tested_colun,NN_model,mean_array,std_array,index,data_matrix):
    nr_short_fiber_tested=nr_intervals

    min_value=data_matrix[:,tested_colun].min()
    max_value=data_matrix[:,tested_colun].max()


    interval_of_values=abs(min_value)+abs(max_value)
    increment_short=(interval_of_values/nr_short_fiber_tested)


    #pego numa linha dos dados de teste, neste caso na linha
    start_array=data_matrix[index]

    #matriz dos dados que vamos usar para fazer a previsão
    start_matrix=np.zeros((nr_short_fiber_tested,15))
    #para ter um array das sequencias
    short_fiber_array=np.zeros(nr_short_fiber_tested)
    real_short_fiber_array=np.zeros(nr_short_fiber_tested)

    i=0
    #meto as linhas da matriz todas iguais ao start array
    while(i<len(start_matrix)):
        start_matrix[i]=start_array
        i+=1



    i=0
    nr_short_fiber=min_value
    while(i<len(short_fiber_array)):
        short_fiber_array[i]=nr_short_fiber
        real_short_fiber_array[i]=nr_short_fiber*std_array[tested_colun]+mean_array[tested_colun]
        nr_short_fiber+=increment_short
        i+=1


    nr_short_fiber=min_value
    nr_short_fiber_i=0
    while(nr_short_fiber_i<nr_short_fiber_tested):
        start_matrix[nr_short_fiber_i,tested_colun]=nr_short_fiber
        nr_short_fiber+=increment_short
        nr_short_fiber_i+=1

    y_pred_fiber = NN_model.predict(start_matrix)
    y_pred_fiber_flat=y_pred_fiber.flatten()



    return(short_fiber_array,y_pred_fiber_flat,real_short_fiber_array)
```

```python
def correlation_simul(min_value,max_value,nr_intervals,tested_colun,NN_model,mean_array,std_array,index,data_matrix):
    nr_short_fiber_tested=nr_intervals

    min_value=data_matrix[:,tested_colun].min()
    max_value=data_matrix[:,tested_colun].max()


    interval_of_values=abs(min_value)+abs(max_value)
    increment_short=(interval_of_values/nr_short_fiber_tested)


    #pego numa linha dos dados de teste, neste caso na linha
    start_array=data_matrix[index]

    #matriz dos dados que vamos usar para fazer a previsão
    start_matrix=np.zeros((nr_short_fiber_tested,15))
    #para ter um array das sequencias
    short_fiber_array=np.zeros(nr_short_fiber_tested)
    real_short_fiber_array=np.zeros(nr_short_fiber_tested)

    i=0
    #meto as linhas da matriz todas iguais ao start array
    while(i<len(start_matrix)):
        start_matrix[i]=start_array
        i+=1



    i=0
    nr_short_fiber=min_value
    while(i<len(short_fiber_array)):
        short_fiber_array[i]=nr_short_fiber
        real_short_fiber_array[i]=nr_short_fiber*std_array[tested_colun]+mean_array[tested_colun]
        nr_short_fiber+=increment_short
        i+=1


    nr_short_fiber=min_value
    nr_short_fiber_i=0
    while(nr_short_fiber_i<nr_short_fiber_tested):
        start_matrix[nr_short_fiber_i,tested_colun]=nr_short_fiber
        nr_short_fiber+=increment_short
        nr_short_fiber_i+=1

    y_pred_fiber = NN_model.predict(start_matrix)
    y_pred_fiber_flat=y_pred_fiber.flatten()



    return(short_fiber_array,y_pred_fiber_flat,real_short_fiber_array)
```

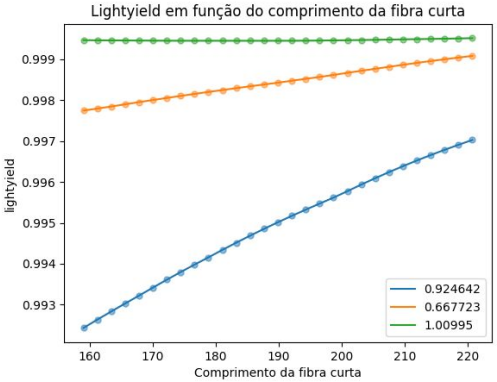
Lightyield depence on fibre size

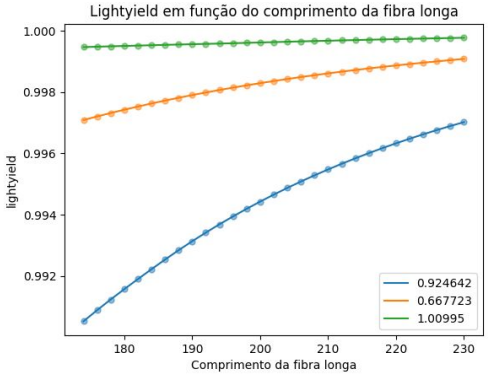
Cell A2





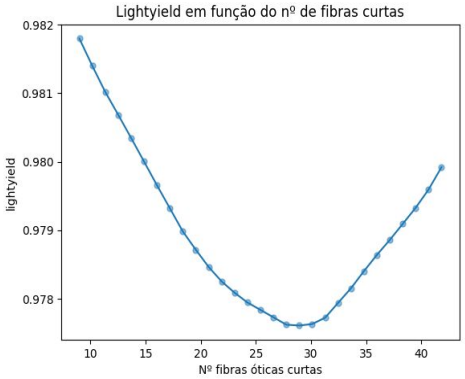
Cell A16

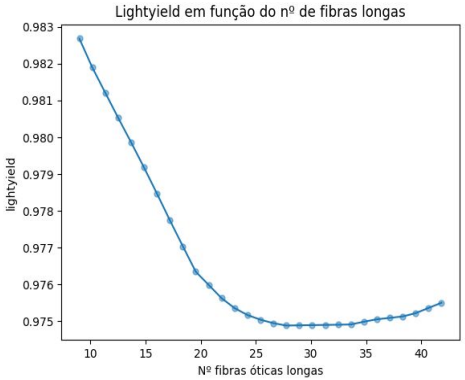


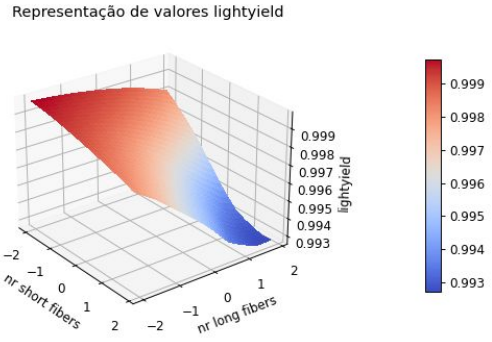


Lightyield depence on the number of fibres

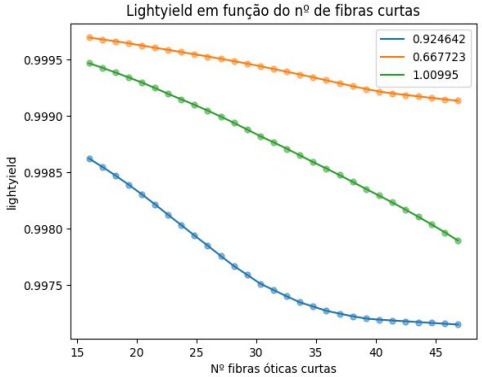
Cell A2

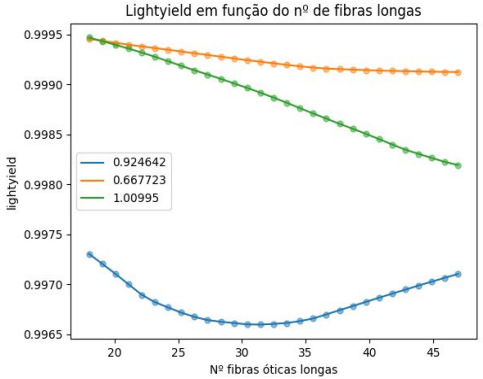






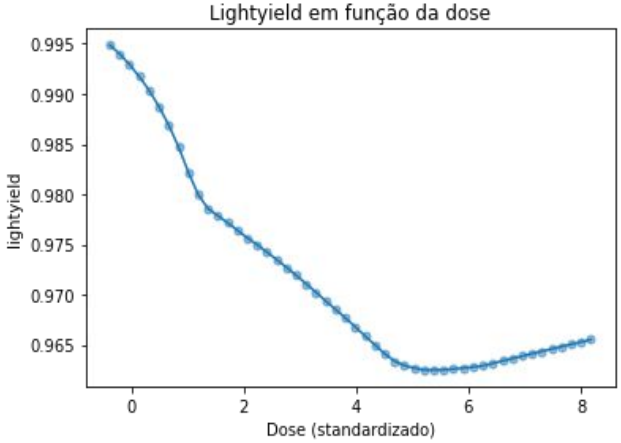
Cell A16

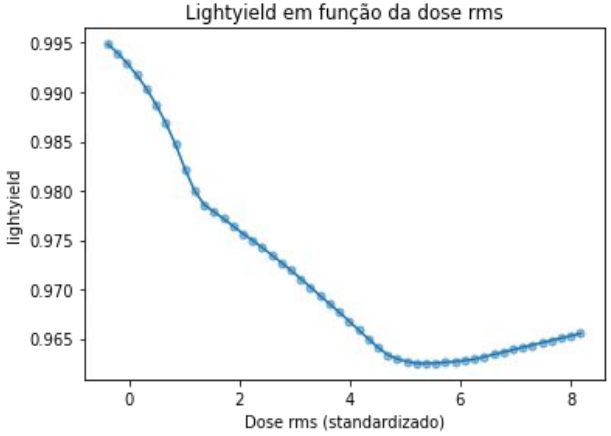




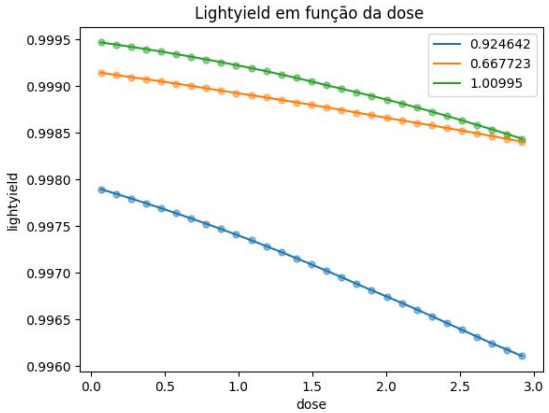
Lightyield depence on dose

Cell A2



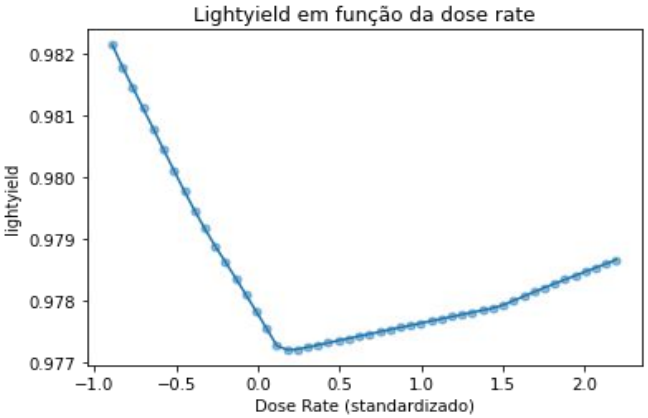


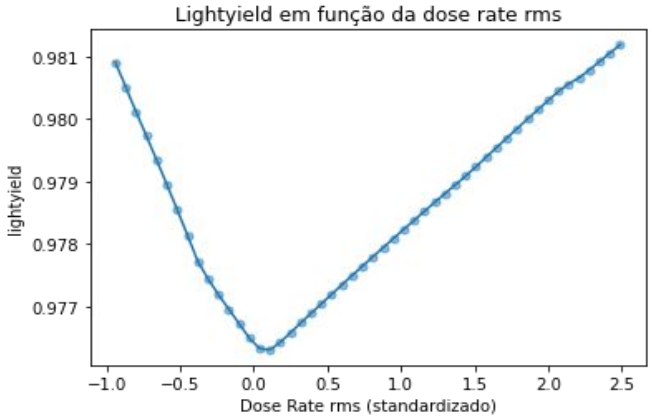
Cell A16



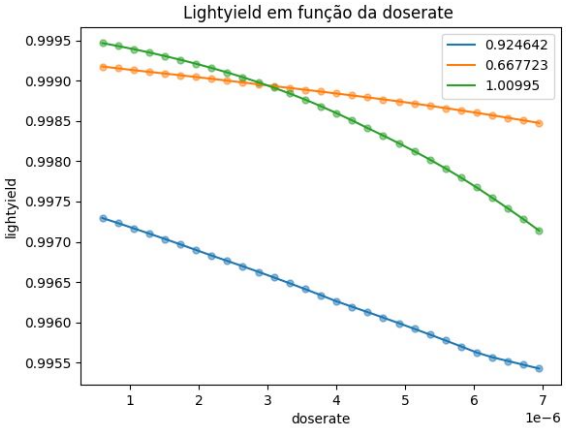
Lightyield depence on dose rate

Cell A2

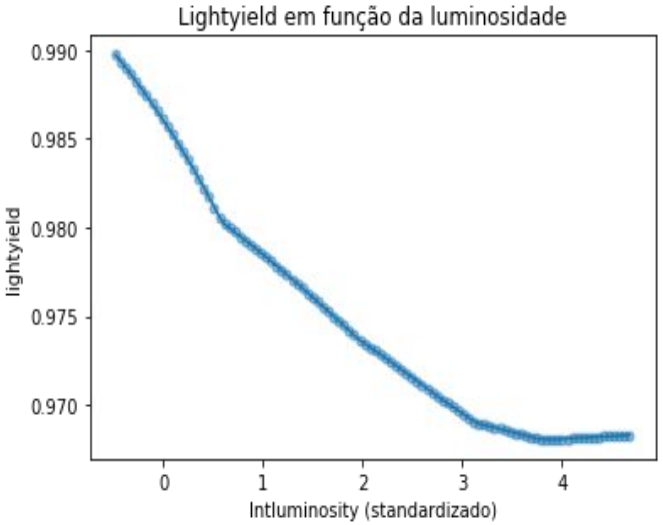




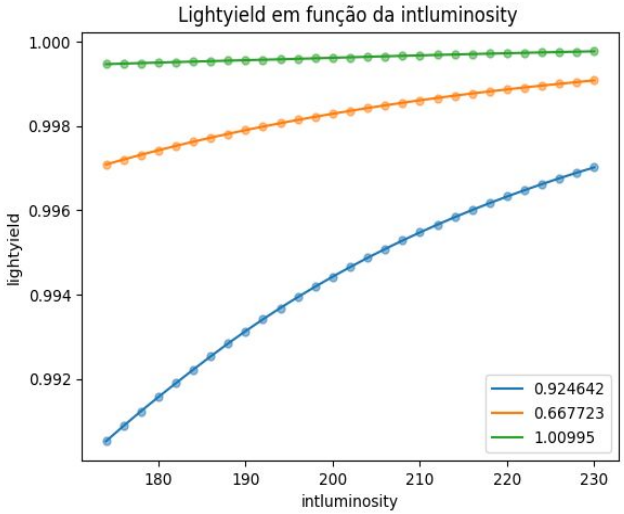
Cell A16



Lightyield depence on luminosity

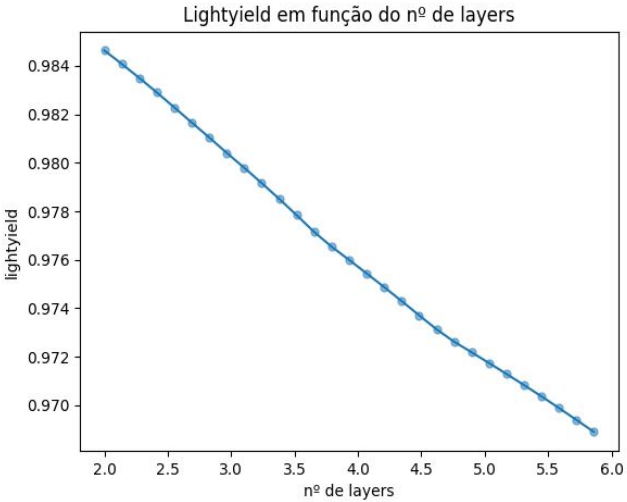


Cell A2

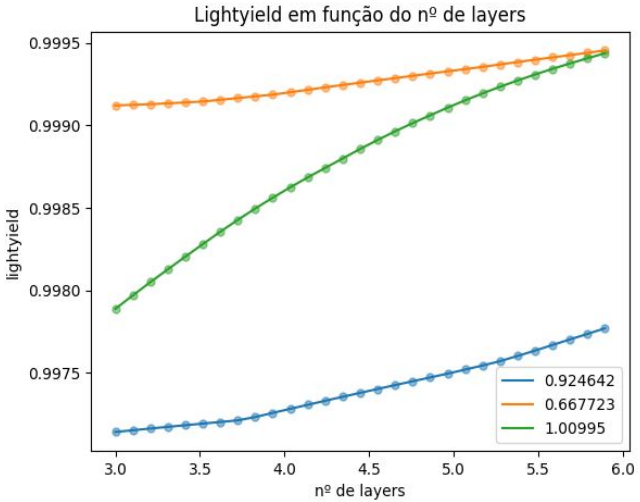


Cell A16

Lightyield depence on the number of layers

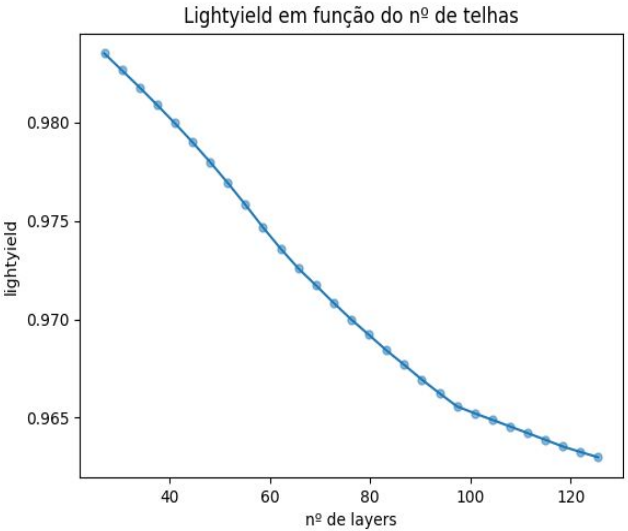


Cell A2

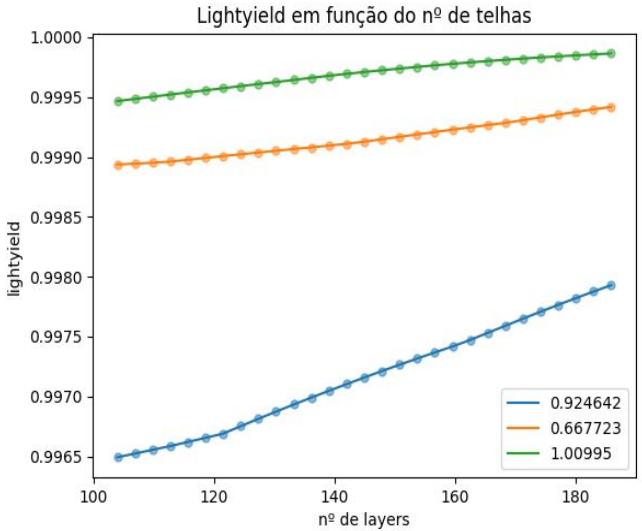


Cell A16

Lightyield depence on the number of tiles



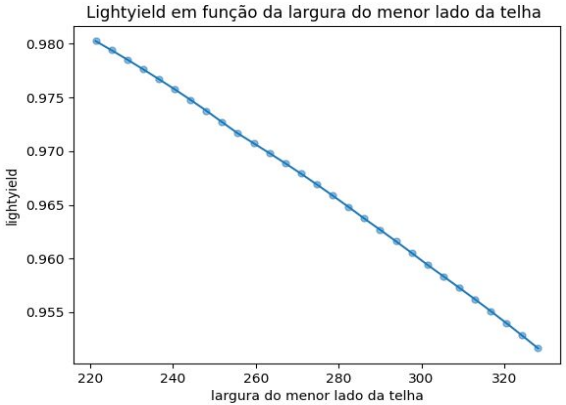
Cell A2

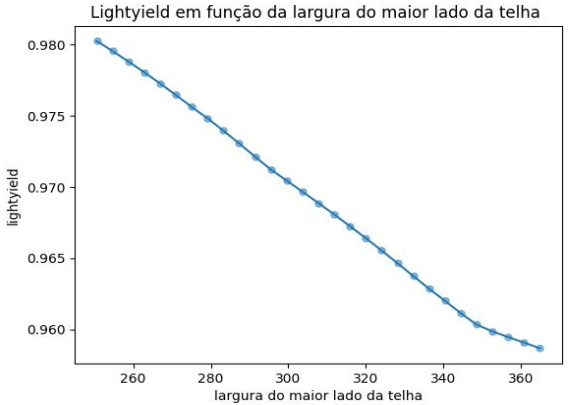


Cell A16

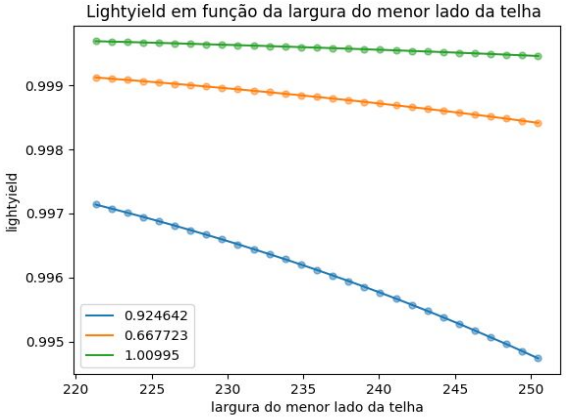
Lightyield depence on tile width

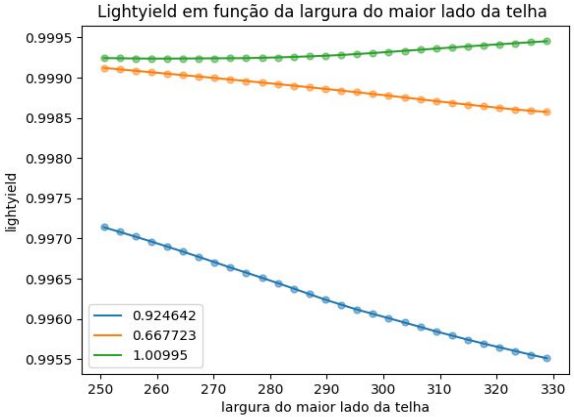
Cell A2



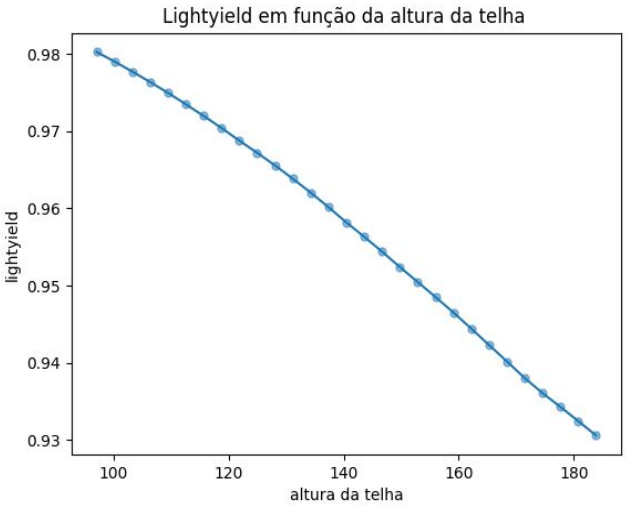


Cell A16

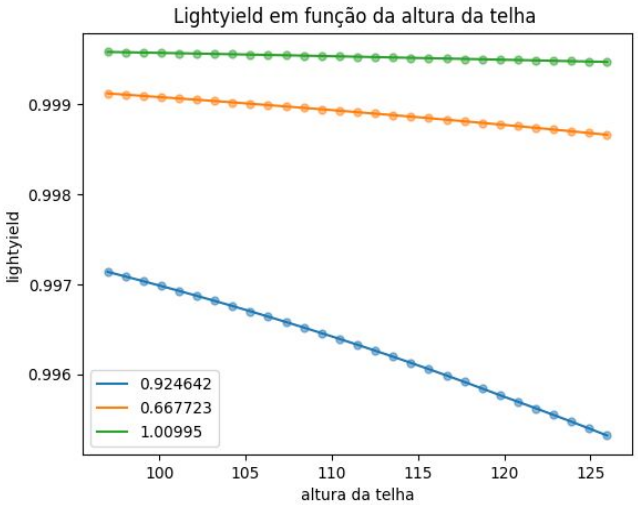




Lightyield depence on tile height

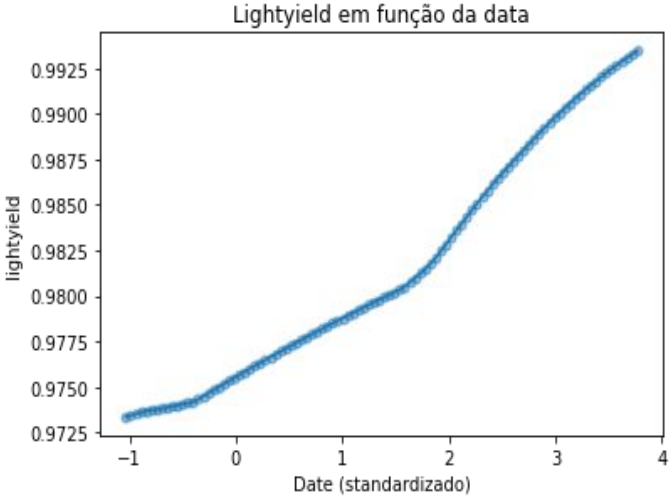


Cell A2

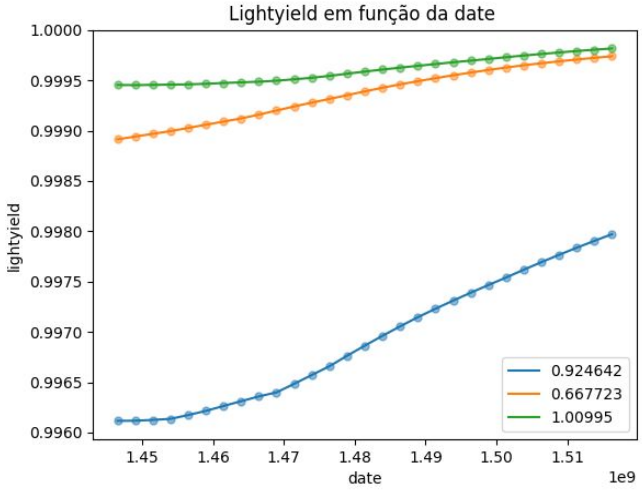


Cell A16

Lightyield depence on date



Cell A2



Cell A16

04

Conclusion

Interpretation of the obtained results

Conclusion

We trained the model with the training dataset in order to predict the value for the lightyield; then we tested with the test dataset and the model was able to predict correctly.

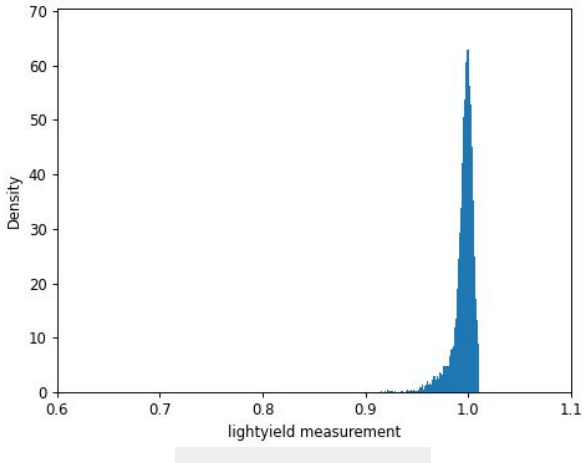
In the real measurements all parameters change simultaneously; therefore, interpreting individual data changes is complex and may not translate reality.

Conclusion

The model had issues predicting certain lightyield values like for the A16 cell and the 0.6 lightyield associated value.

There is a lack of measurements in that range, the model was not able to predict them correctly.

A further study on the range of predictions of the model needs to be done.



References

[1] J. Abdallah et al, *The optical instrumentation of the ATLAS Tile Calorimeter*, 2013 JINST 8 P01005
[2] Merve Nazlim Agaras, *The ATLAS Tile Calorimeter performance and its upgrade towards the High-Luminosity LHC*, The ATLAS collaboration, january 2022

Questions?

Thank you!