Atividade 3

Lucas Roda Ximenes dos Santos
(11917239)

Instituto de Física, Universidade de São Paulo, Rua do Matão 1371, 05508-090, Cidade Universitária, São Paulo, Brasil

(Data: 16 de novembro de 2025)

O objetivo dessa atividade é criar uma rede neural que possa ser treinada, usando apenas as 21 low-level features, para separar o sinal do fundo.

Comentário: Em um repositório no Github salvei um arquivo nomeado `Atividade3.ipynb`, contendo as mesmas informações e comentários presentes neste PDF, então para tornar o código verificável, os arquivos estão contidos nos *shields* abaixo, basta clicar em qual arquivo é mais simples de verificar.

1 Análise exploratória

Vamos primeiro importar os dados que iremos utilizar. Como o arquivo que contém os dados é relativamente grande (69.6 MB), ele foi adicionado no drive e importado no Google Colab usando a biblioteca `gdown`. Os dados originais serão armazenados na variável `dados`.

```
import gdown
import pandas as pd

file = {
    'filename': 'HIGGS_100k.csv',
    "file_id" : "1mrzlmpPh4copC7ZULocAGhA1xAep88Sy"
}

url = f"https://drive.google.com/uc?id={file['file_id']}"
gdown.download(url, file['filename'], quiet=True)
print(f'{file['filename']} baixado com sucesso e armazenado em "dados".')

dados = pd.read_csv('HIGGS_100k.csv')
```

```
HIGGS_100k.csv baixado com sucesso e armazenado em "dados".
```

Antes de começar, vamos renomear os atributos apropriadamente, dado que eles estão originalmente nomeados como números, o que torna a análise mais complicada. Como informado na proposta da atividade, as colunas em ordem são: `class label`, `lepton pT`, `lepton eta`, `lepton phi`, `missing energy magnitude`, `missing energy phi`, `jet 1 pt`, `jet 1 eta`, `jet 1 phi`, `jet 1 b-tag`, `jet 2 pt`, `jet 2 eta`, `jet 2 phi`, `jet 2 b-tag`, `jet 3 pt`,

`jet 3 eta`, `jet 3 phi`, `jet 3 b-tag`, `jet 4 pt`, `jet 4 eta`, `jet 4 phi`, `jet 4 b-tag`, `m_jj`, `m_jjj`, `m_lv`, `m_jlv`, `m_bb`, `m_wbb` e `m_wwbb`, portanto serão estes os nomes atribuidos a cada variável.

```
names = {
'1.0000000000000000e+00': 'class label',
'8.692932128906250000e-01': 'lepton pT',
'-6.350818276405334473e-01': 'lepton eta',
'2.256902605295181274e-01': 'lepton phi',
'3.274700641632080078e-01': 'missing energy magnitude',
'-6.899932026863098145e-01': 'missing energy phi',
'7.542022466659545898e-01': 'jet 1 pt',
'-2.485731393098831177e-01': 'jet 1 eta',
'-1.092063903808593750e+00': 'jet 1 phi',
'0.000000000000000000e+00': 'jet 1 b-tag',
'1.374992132186889648e+00': 'jet 2 pt',
'-6.536741852760314941e-01': 'jet 2 eta',
'9.303491115570068359e-01': 'jet 2 phi',
'1.107436060905456543e+00': 'jet 2 b-tag',
'1.138904333114624023e+00': 'jet 3 pt',
'-1.578198313713073730e+00': 'jet 3 eta',
'-1.046985387802124023e+00': 'jet 3 phi',
'0.000000000000000000e+00.1': 'jet 3 b-tag',
'6.579295396804809570e-01': 'jet 4 pt',
'-1.045456994324922562e-02': 'jet 4 eta',
'-4.576716944575309753e-02': 'jet 4 phi',
'3.101961374282836914e+00': 'jet 4 b-tag',
'1.353760004043579102e+00': 'm_jj',
'9.795631170272827148e-01': 'm_jjj',
'9.780761599540710449e-01': 'm_lv',
'9.200048446655273438e-01': 'm_jlv',
'7.216574549674987793e-01': 'm_bb',
'9.887509346008300781e-01': 'm_wbb',
'8.766783475875854492e-01': 'm_wwbb',
}

dados.rename(columns=names, inplace=True)
```

Como procedimento padrão, verifiquemos se existem dados faltantes e analisemos as características principais de cada um dos atributos.

```
dados.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 100000 entries, 0 to 99999
Data columns (total 29 columns):
#   Column                    Non-Null Count   Dtype
---  ------                    --------------   -----
0   class label               100000 non-null  float64
1   lepton pT                 100000 non-null  float64
2   lepton eta                100000 non-null  float64
3   lepton phi                100000 non-null  float64
```

```
4   missing energy magnitude  100000 non-null  float64
5   missing energy phi        100000 non-null  float64
6   jet 1 pt                  100000 non-null  float64
7   jet 1 eta                 100000 non-null  float64
8   jet 1 phi                 100000 non-null  float64
9   jet 1 b-tag               100000 non-null  float64
10  jet 2 pt                  100000 non-null  float64
11  jet 2 eta                 100000 non-null  float64
12  jet 2 phi                 100000 non-null  float64
13  jet 2 b-tag               100000 non-null  float64
14  jet 3 pt                  100000 non-null  float64
15  jet 3 eta                 100000 non-null  float64
16  jet 3 phi                 100000 non-null  float64
17  jet 3 b-tag               100000 non-null  float64
18  jet 4 pt                  100000 non-null  float64
19  jet 4 eta                 100000 non-null  float64
20  jet 4 phi                 100000 non-null  float64
21  jet 4 b-tag               100000 non-null  float64
22  m_jj                      100000 non-null  float64
23  m_jjj                     100000 non-null  float64
24  m_lv                      100000 non-null  float64
25  m_jlv                     100000 non-null  float64
26  m_bb                      100000 non-null  float64
27  m_wbb                     100000 non-null  float64
28  m_wwbb                    100000 non-null  float64
dtypes: float64(29)
memory usage: 22.1 MB
```

```
dados.describe()
```

	class label	lepton pT	lepton eta	···	m_bb	m_wbb	m_wwbb
count	100000.0000	100000.0000	100000.0000	···	100000.0000	100000.0000	100000.0000
mean	0.528330	0.990366	-0.003806	···	0.973076	1.031874	0.959203
std	0.499199	0.561840	1.004840	···	0.523557	0.363395	0.313258
min	0.000000	0.274697	-2.434976	···	0.048125	0.303350	0.350939
25%	0.000000	0.590936	-0.741244	···	0.673789	0.819170	0.769964
50%	1.000000	0.854835	-0.002976	···	0.874004	0.947037	0.871038
75%	1.000000	1.236776	0.735292	···	1.139816	1.139032	1.057479
max	1.000000	7.805887	2.433894	···	11.994177	7.318191	6.015647

É evidente que todos os dados estão em uma mesma escala de valores, então a priori não será necessário nenhum tipo de reescalonamento. O foco é utilizar apenas os atributos entitulados *low-level*, então separaremos os 29 atributos

em 3 conjuntos: o que contém apenas a variável alvo, que é a `class label`, o que contém os 21 atributos *low-level* e os 7 restantes que são os atributos *high-level*.

```
classLabel = dados.iloc[:, 0]
lowLevel = dados.iloc[:, 1:22]
highLevel = dados.iloc[:, 22:29]
```

Como são muitos dados (100.000), é interessante verificar se eles estão balanceados, isto é, se a quantidade de dados com `class label == 1` e semelhante à quantidade de dados com `class label == 0`.

```
print("Distribuição das classes:")
print(classLabel.value_counts(normalize=True) * 100)  # Em percentual
```

```
Distribuição das classes:
class label
1.0    52.833
0.0    47.167
Name: proportion, dtype: float64
```

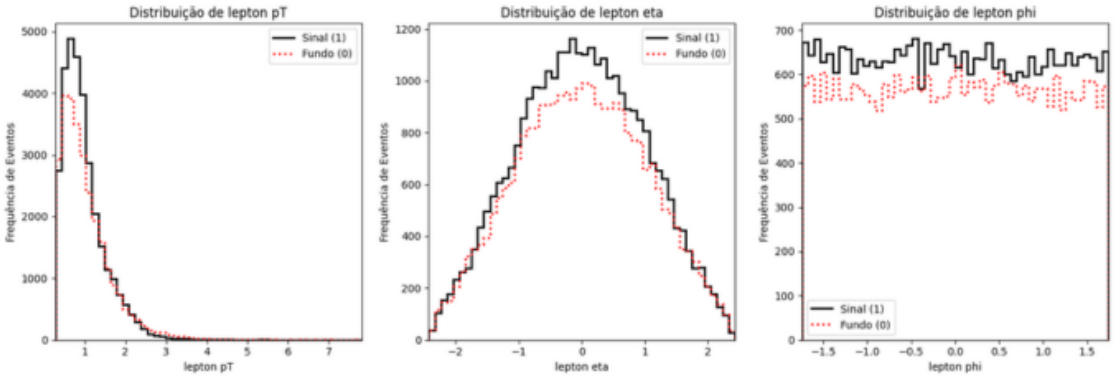
Temos então um conjunto de dados bem balanceado, como esperado, dado que estes foram utilizados no artigo original, com $\approx 53\%$ dos dados classificados como sinal (`class label == 1`) e $\approx 47\%$ classificados como fundo (`class label == 0`). Podemos então separar os dados em conjuntos de treino, validação e teste para treinar a rede neural. Faremos uma separação de 60% para treino, validação 15% e 25% para teste. E para reprodutibilidade, utilizaremos um `random_state == 42`.

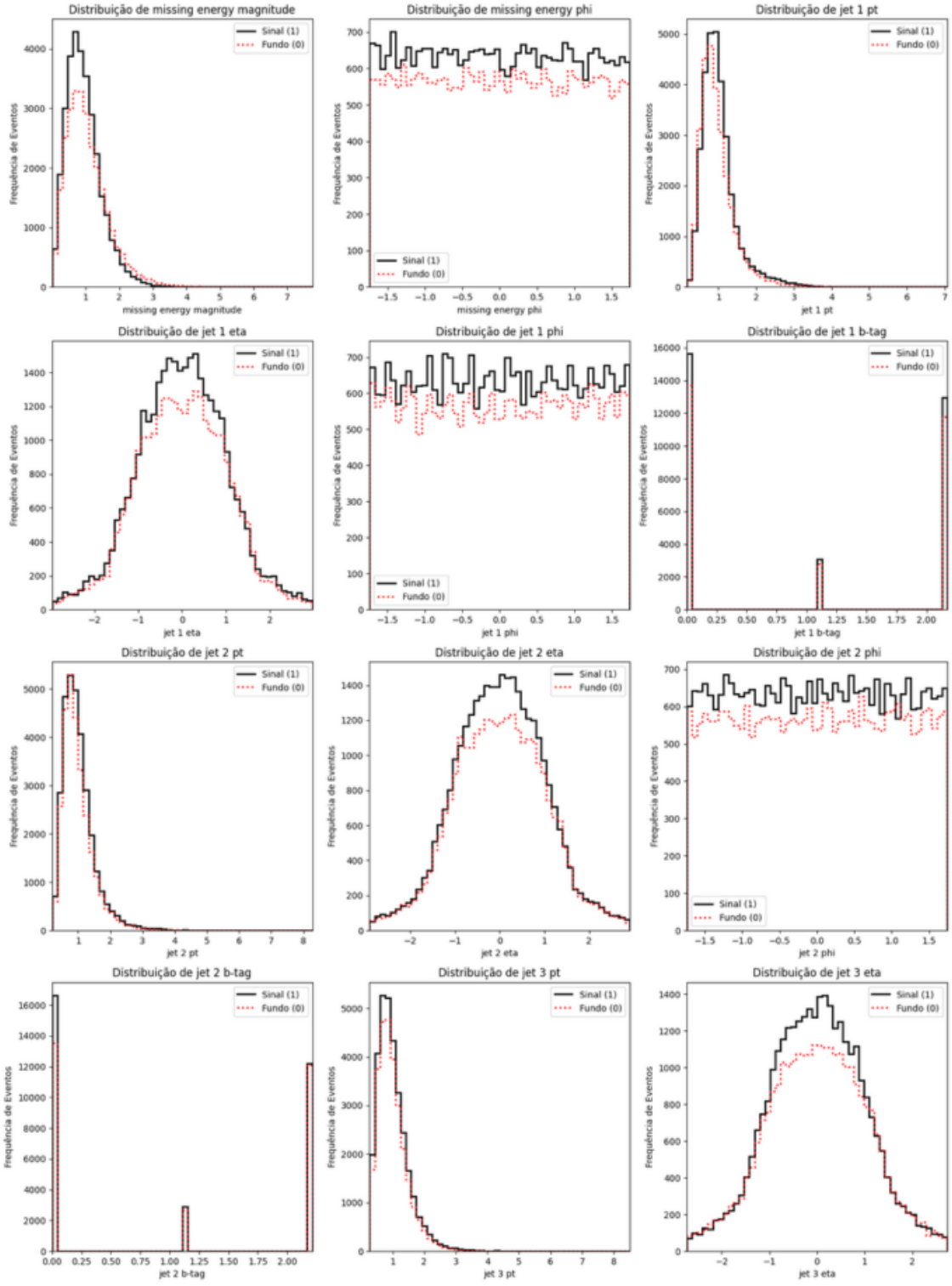
```
from sklearn.model_selection import train_test_split
import numpy as np

trainRatio = 0.6
validationRatio = 0.15
testRatio = 0.25

randomState = 42

X = lowLevel
y = classLabel

XTrain, XTest, yTrain, yTest = train_test_split(X, y, test_size = 1 - trainRatio, stratify = y,
↪  random_state = randomState)
XVal, XTest, yVal, yTest = train_test_split(XTest, yTest, test_size = testRatio/(testRatio +
↪  validationRatio), stratify = yTest, random_state = randomState)
```

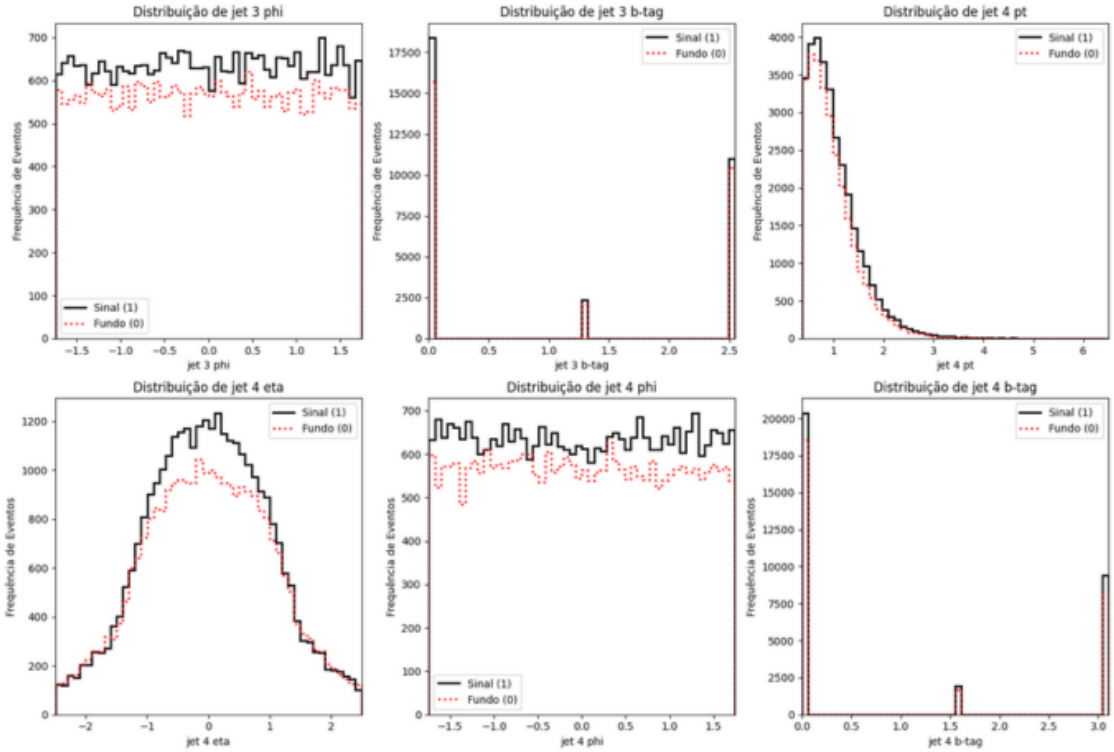
Uma verificação importante a se fazer com os dados é sobre o comportamento dos atributos em relação às ocorrências, isto é, a frequência de eventos. Para o conjunto de dados de treino que iremos utilizar (`XTrain`), podemos fazer o plot dos 21 atributos *low-level* com base na quantidade de dados separados em sinal e fundo, e verificar o comportamento destes em relação à frequência dos eventos.

```
import matplotlib.pyplot as plt
from matplotlib.lines import Line2D
```

```
lowLevelWL = XTrain.copy()
lowLevelWL['class label'] = classLabel

fig, axs = plt.subplots(7, 3, figsize=(15, 35))
axs = axs.flatten()

for i, feature in enumerate(lowLevel.columns):
    data1 = lowLevelWL[lowLevelWL['class label'] == 1][feature]
    data0 = lowLevelWL[lowLevelWL['class label'] == 0][feature]

    min_all = min(data1.min(), data0.min())
    max_all = max(data1.max(), data0.max())

    bins = np.linspace(min_all, max_all, 51)

    axs[i].hist(data1, histtype='step', linewidth=2, color='black', label='Sinal (1)', bins=bins,
    ↪  density=False)
    axs[i].hist(data0, histtype='step', linewidth=2, color='red', linestyle='dotted', label='Fundo (0)',
    ↪  bins=bins, density=False)

    axs[i].set_title(f'Distribuição de {feature}')
    axs[i].set_xlabel(feature)
    axs[i].set_ylabel('Frequência de Eventos')
    axs[i].legend()

    legend_elements = [
        Line2D([0], [0], color='black', linewidth=2, label='Sinal (1)'),
        Line2D([0], [0], color='red', linewidth=2, linestyle='dotted', label='Fundo (0)')
    ]
    axs[i].legend(handles=legend_elements)

    axs[i].set_xlim(min_all, max_all)

plt.tight_layout()
plt.show()
```




Distribuição de missing energy magnitude
Distribuição de missing energy phi
Distribuição de jet 1 pt
Distribuição de jet 1 eta
Distribuição de jet 1 phi
Distribuição de jet 1 b-tag
Distribuição de jet 2 pt
Distribuição de jet 2 eta
Distribuição de jet 2 phi
Distribuição de jet 2 b-tag
Distribuição de jet 3 pt
Distribuição de jet 3 eta
Sinal (1)
Fundo (0)
Frequência de Eventos
missing energy magnitude
missing energy phi
jet 1 pt
jet 1 eta
jet 1 phi
jet 1 b-tag
jet 2 pt
jet 2 eta
jet 2 phi
jet 2 b-tag
jet 3 pt
jet 3 eta

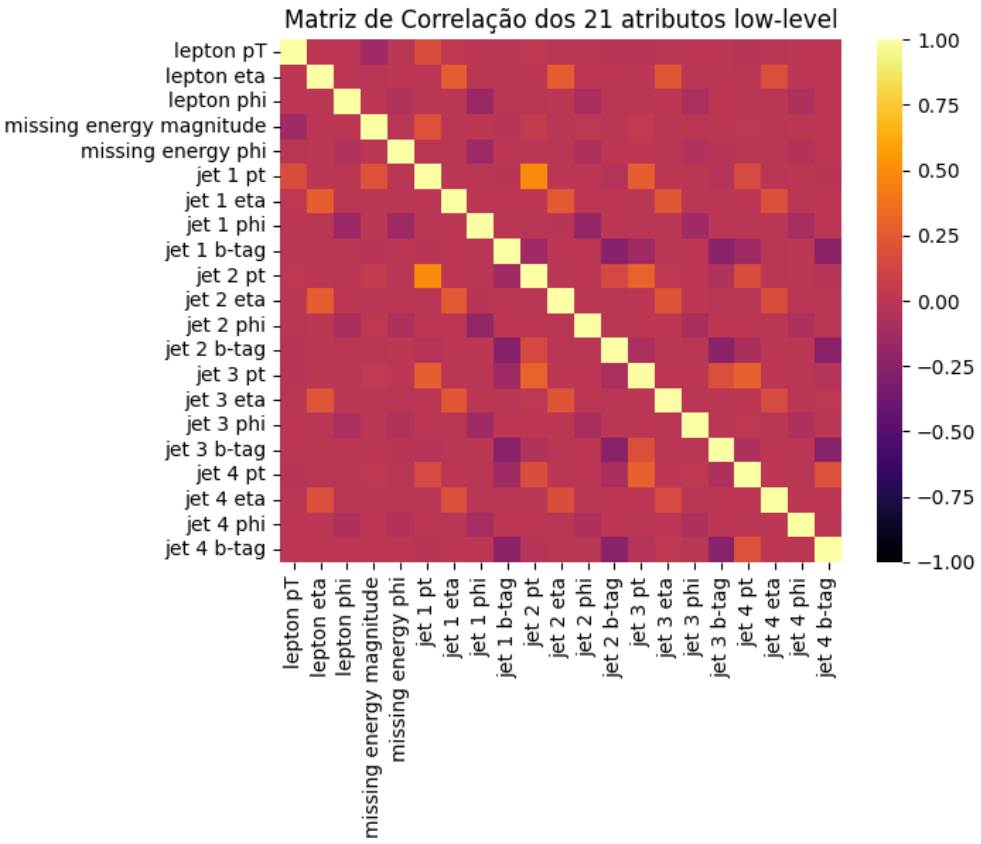


Podemos ver então alguns comportamentos que se repetem em relação a alguns atributos (o que é de certa forma esperado, dado o nome de alguns deles). É possível então separar os atributos em 4 tipos de comportamento:

- Comportamento similar ao de uma curva da forma axe^{-x}, com a constante (não necessariamente essa função, mas muito similar):
  - `lepton pT`, `missing energy magnitude`, `jet 1 pt`, `jet 2 pt`, `jet 3 pt` e `jet 4 pt`;
- Comportamento mais uniforme uniforme, similar a uma curva $y = a$, com a constante:
  - `lepton phi`, `missing energy phi`, `jet 1 phi`, `jet 2 phi`, `jet 3 phi` e `jet 4 phi`;
- Comportamento similar a uma gaussiana ae^{-x^2}, com a constante:
  - `lepton eta`, `jet 1 eta`, `jet 2 eta`, `jet 3 eta` e `jet 4 eta`;
- Comportamento de ”picos”:
  - `jet 1 b-tag`, `jet 2 b-tag`, `jet 3 b-tag` e `jet 4 b-tag`.

Estes comportamentos distintos implicam em cinemáticas distintas do conjunto de dados, permitindo mais informações físicas a serem interpretadas. É interessante agora verificar se os atributos em questão possuem alguma correlação forte, para caso exista, lidarmos com isto (mesmo que neste caso a correlação é inesperada, dados que os atributos são atributos físicos independentes).

```
import seaborn as sns

corr_matrix = XTrain.corr()

sns.heatmap(corr_matrix, annot=False, cmap='inferno', vmin=-1, vmax=1)
plt.title('Matriz de Correlação dos 21 atributos low-level')
plt.show()
```



Vemos então, conforme o esperado, que os atributos não possuem nenhuma correlação muito forte, o que faz com que não precisemos lidar com esse tipo de problema. Por fim, podemos verificar a presença de *outliers* no conjunto de dados completo (se não houver no conjunto todo, logo não há nos conjuntos de treino, validação e teste). Consideraremos valores potencialmente descritos como *outliers* aqueles cujo módulo da diferença entre o valor x e a média μ forem maior do que 3 vezes o desvio padrão do atributo, isto é $|x-\mu| > 3\sigma$.

```
lowDesc = lowLevel.describe()
outliers = (lowLevel > lowDesc.loc['mean'] + 3 * lowDesc.loc['std']) | (lowLevel < lowDesc.loc['mean'] - 3
↪  * lowDesc.loc['std'])
```

```
print("Contagem de outliers potenciais por atributo:")
print(outliers.sum())
```

```
Contagem de outliers potenciais por atributo:
lepton pT                   1480
lepton eta                     0
lepton phi                     0
missing energy magnitude    1279
missing energy phi             0
jet 1 pt                    1965
jet 1 eta                      0
jet 1 phi                      0
jet 1 b-tag                    0
jet 2 pt                    1615
jet 2 eta                      0
jet 2 phi                      0
jet 2 b-tag                    0
jet 3 pt                    1388
jet 3 eta                      0
jet 3 phi                      0
jet 3 b-tag                    0
jet 4 pt                    1481
jet 4 eta                      0
jet 4 phi                      0
jet 4 b-tag                    0
dtype: int64
```

Levando em consideração que o conjunto de dados inteiro possui 100.000 dados, a quantidade de valores que podem possivelmente ser *outliers* é muito menor, então podemos desconsiderar o tratamento de *outliers* nos dados.

2 Construção da rede neural

Feita a pré-análise, podemos partir para criação da rede neural e seu treinamento. O pacote escolhido para isso vai ser o `Keras`. Feitos alguns testes com algumas redes de **perceptrons**, a melhor opção foi uma rede de 5 camadas, onde 3 delas vão ter 64 **perceptrons** com função de ativação `relu`, uma vai ter 32 **perceptrons**, também com ativação `relu` e uma camada final com um único **perceptron** com ativação `sigmoid`. Para compilar, utilizaremos a opção de otimizador `adam`, a função perda `binary_crossentropy`, dado que nosso alvo consiste de resultados 0 ou 1 e a métrica vai ser de acurácia. A fim de evitar *overfitting*, usaremos um `callback` definido pela função `EarlyStopping`, que vai interromper o treinamento quando a perda voltar a subir ao invés de continuar decrescendo. Utilizaremos também 50 `epochs` e um `batch_size` de 32, para manter a rede o mais simples possível.

```
import tensorflow as tf
from tensorflow import keras
from tensorflow.keras import layers
from tensorflow.keras.callbacks import EarlyStopping
```

```
# Modelo simples MLP
model = keras.Sequential([
    layers.Input(shape=(len(lowLevel.columns),)),
    layers.Dense(64, activation='relu'),
    layers.Dropout(0.2),
    layers.Dense(64, activation='relu'),
    layers.Dropout(0.2),
    layers.Dense(64, activation='relu'),
    layers.Dropout(0.2),
    layers.Dense(32, activation='relu'),
    layers.Dense(1, activation='sigmoid')
])

model.compile(optimizer='adam', loss='binary_crossentropy', metrics=['accuracy'])

earlyStop = EarlyStopping(monitor='val_loss', patience=5, restore_best_weights=True)

history = model.fit(XTrain, yTrain, epochs=50, batch_size=32, validation_data=(XVal, yVal),
↪  callbacks=[earlyStop], verbose=1)
```

```
Epoch 1/50
1875/1875 [          ] 10s 4ms/step - accuracy: 0.5333 - loss: 0.6917 - val_accuracy: 0.5763 - val_loss:
↪  0.6721
Epoch 2/50
1875/1875 [          ] 8s 4ms/step - accuracy: 0.5735 - loss: 0.6745 - val_accuracy: 0.5960 - val_loss:
↪  0.6633
Epoch 3/50
1875/1875 [          ] 8s 3ms/step - accuracy: 0.5918 - loss: 0.6676 - val_accuracy: 0.6010 - val_loss:
↪  0.6593
Epoch 4/50
1875/1875 [          ] 5s 3ms/step - accuracy: 0.6020 - loss: 0.6610 - val_accuracy: 0.6092 - val_loss:
↪  0.6534
Epoch 5/50
1875/1875 [          ] 5s 3ms/step - accuracy: 0.6061 - loss: 0.6554 - val_accuracy: 0.6113 - val_loss:
↪  0.6533
Epoch 6/50
1875/1875 [          ] 6s 3ms/step - accuracy: 0.6138 - loss: 0.6533 - val_accuracy: 0.6143 - val_loss:
↪  0.6564
Epoch 7/50
1875/1875 [          ] 5s 3ms/step - accuracy: 0.6121 - loss: 0.6516 - val_accuracy: 0.6187 - val_loss:
↪  0.6493
Epoch 8/50
1875/1875 [          ] 6s 3ms/step - accuracy: 0.6133 - loss: 0.6504 - val_accuracy: 0.6173 - val_loss:
↪  0.6463
Epoch 9/50
1875/1875 [          ] 10s 3ms/step - accuracy: 0.6230 - loss: 0.6469 - val_accuracy: 0.6192 - val_loss:
↪  0.6454
Epoch 10/50
1875/1875 [          ] 5s 3ms/step - accuracy: 0.6257 - loss: 0.6442 - val_accuracy: 0.6297 - val_loss:
↪  0.6419
Epoch 11/50
1875/1875 [          ] 5s 3ms/step - accuracy: 0.6225 - loss: 0.6448 - val_accuracy: 0.6276 - val_loss:
↪  0.6420
Epoch 12/50
1875/1875 [          ] 7s 3ms/step - accuracy: 0.6281 - loss: 0.6404 - val_accuracy: 0.6312 - val_loss:
↪  0.6423
```

```
Epoch 13/50
1875/1875 [==============================] 9s 3ms/step - accuracy: 0.6263 - loss: 0.6413 - val_accuracy: 0.6251 - val_loss:
↪  0.6424
Epoch 14/50
1875/1875 [==============================] 5s 3ms/step - accuracy: 0.6365 - loss: 0.6368 - val_accuracy: 0.6323 - val_loss:
↪  0.6388
Epoch 15/50
1875/1875 [==============================] 11s 3ms/step - accuracy: 0.6330 - loss: 0.6366 - val_accuracy: 0.6291 - val_loss:
↪  0.6382
Epoch 16/50
1875/1875 [==============================] 5s 3ms/step - accuracy: 0.6308 - loss: 0.6381 - val_accuracy: 0.6337 - val_loss:
↪  0.6350
Epoch 17/50
1875/1875 [==============================] 5s 3ms/step - accuracy: 0.6369 - loss: 0.6345 - val_accuracy: 0.6354 - val_loss:
↪  0.6355
Epoch 18/50
1875/1875 [==============================] 5s 3ms/step - accuracy: 0.6320 - loss: 0.6374 - val_accuracy: 0.6370 - val_loss:
↪  0.6343
Epoch 19/50
1875/1875 [==============================] 5s 2ms/step - accuracy: 0.6364 - loss: 0.6338 - val_accuracy: 0.6385 - val_loss:
↪  0.6334
Epoch 20/50
1875/1875 [==============================] 6s 3ms/step - accuracy: 0.6354 - loss: 0.6345 - val_accuracy: 0.6381 - val_loss:
↪  0.6327
Epoch 21/50
1875/1875 [==============================] 10s 3ms/step - accuracy: 0.6409 - loss: 0.6319 - val_accuracy: 0.6398 - val_loss:
↪  0.6315
Epoch 22/50
1875/1875 [==============================] 5s 3ms/step - accuracy: 0.6360 - loss: 0.6318 - val_accuracy: 0.6377 - val_loss:
↪  0.6319
Epoch 23/50
1875/1875 [==============================] 5s 3ms/step - accuracy: 0.6395 - loss: 0.6315 - val_accuracy: 0.6397 - val_loss:
↪  0.6340
Epoch 24/50
1875/1875 [==============================] 6s 3ms/step - accuracy: 0.6421 - loss: 0.6289 - val_accuracy: 0.6375 - val_loss:
↪  0.6310
Epoch 25/50
1875/1875 [==============================] 8s 4ms/step - accuracy: 0.6437 - loss: 0.6279 - val_accuracy: 0.6354 - val_loss:
↪  0.6336
Epoch 26/50
1875/1875 [==============================] 6s 3ms/step - accuracy: 0.6464 - loss: 0.6266 - val_accuracy: 0.6411 - val_loss:
↪  0.6304
Epoch 27/50
1875/1875 [==============================] 5s 3ms/step - accuracy: 0.6448 - loss: 0.6275 - val_accuracy: 0.6425 - val_loss:
↪  0.6288
Epoch 28/50
1875/1875 [==============================] 6s 3ms/step - accuracy: 0.6463 - loss: 0.6262 - val_accuracy: 0.6448 - val_loss:
↪  0.6291
Epoch 29/50
1875/1875 [==============================] 5s 3ms/step - accuracy: 0.6468 - loss: 0.6264 - val_accuracy: 0.6431 - val_loss:
↪  0.6299
Epoch 30/50
1875/1875 [==============================] 6s 3ms/step - accuracy: 0.6436 - loss: 0.6282 - val_accuracy: 0.6445 - val_loss:
↪  0.6294
Epoch 31/50
1875/1875 [==============================] 5s 3ms/step - accuracy: 0.6417 - loss: 0.6268 - val_accuracy: 0.6408 - val_loss:
↪  0.6280
Epoch 32/50
```

```
1875/1875 [==============] 5s 3ms/step - accuracy: 0.6482 - loss: 0.6244 - val_accuracy: 0.6473 - val_loss:
↪  0.6275
Epoch 33/50
1875/1875 [==============] 6s 3ms/step - accuracy: 0.6480 - loss: 0.6222 - val_accuracy: 0.6435 - val_loss:
↪  0.6273
Epoch 34/50
1875/1875 [==============] 5s 3ms/step - accuracy: 0.6498 - loss: 0.6222 - val_accuracy: 0.6392 - val_loss:
↪  0.6288
Epoch 35/50
1875/1875 [==============] 6s 3ms/step - accuracy: 0.6481 - loss: 0.6238 - val_accuracy: 0.6415 - val_loss:
↪  0.6285
Epoch 36/50
1875/1875 [==============] 5s 3ms/step - accuracy: 0.6530 - loss: 0.6224 - val_accuracy: 0.6436 - val_loss:
↪  0.6260
Epoch 37/50
1875/1875 [==============] 6s 3ms/step - accuracy: 0.6495 - loss: 0.6221 - val_accuracy: 0.6472 - val_loss:
↪  0.6272
Epoch 38/50
1875/1875 [==============] 10s 3ms/step - accuracy: 0.6486 - loss: 0.6239 - val_accuracy: 0.6436 - val_loss:
↪  0.6263
Epoch 39/50
1875/1875 [==============] 6s 3ms/step - accuracy: 0.6542 - loss: 0.6179 - val_accuracy: 0.6477 - val_loss:
↪  0.6257
Epoch 40/50
1875/1875 [==============] 10s 3ms/step - accuracy: 0.6509 - loss: 0.6219 - val_accuracy: 0.6467 - val_loss:
↪  0.6243
Epoch 41/50
1875/1875 [==============] 5s 3ms/step - accuracy: 0.6511 - loss: 0.6211 - val_accuracy: 0.6486 - val_loss:
↪  0.6243
Epoch 42/50
1875/1875 [==============] 5s 3ms/step - accuracy: 0.6528 - loss: 0.6202 - val_accuracy: 0.6483 - val_loss:
↪  0.6254
Epoch 43/50
1875/1875 [==============] 6s 3ms/step - accuracy: 0.6562 - loss: 0.6185 - val_accuracy: 0.6485 - val_loss:
↪  0.6240
Epoch 44/50
1875/1875 [==============] 5s 3ms/step - accuracy: 0.6537 - loss: 0.6174 - val_accuracy: 0.6478 - val_loss:
↪  0.6248
Epoch 45/50
1875/1875 [==============] 6s 3ms/step - accuracy: 0.6537 - loss: 0.6178 - val_accuracy: 0.6499 - val_loss:
↪  0.6250
Epoch 46/50
1875/1875 [==============] 5s 3ms/step - accuracy: 0.6510 - loss: 0.6224 - val_accuracy: 0.6495 - val_loss:
↪  0.6245
Epoch 47/50
1875/1875 [==============] 6s 3ms/step - accuracy: 0.6562 - loss: 0.6168 - val_accuracy: 0.6489 - val_loss:
↪  0.6257
Epoch 48/50
1875/1875 [==============] 6s 3ms/step - accuracy: 0.6547 - loss: 0.6184 - val_accuracy: 0.6502 - val_loss:
↪  0.6241
```

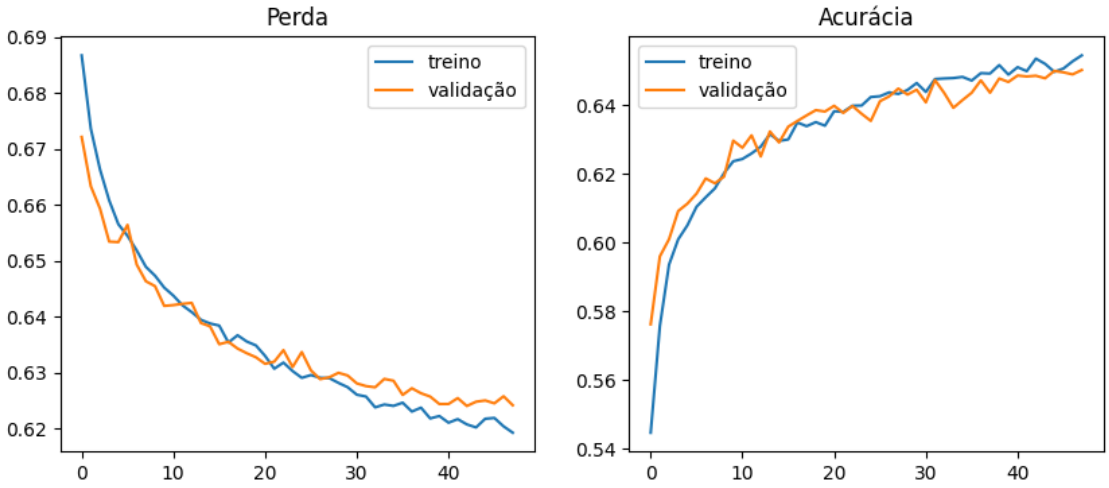
Feito o treinamento da rede, podemos salvar a acurácia final obtida.

```
test_loss, test_acc = model.evaluate(XTest, yTest)
print(f'Acurácia no teste: {test_acc:.4f}')
```

```
782/782 ━━━━━━━━━━━━━━━━━━━━ 1s 2ms/step - accuracy: 0.6439 - loss: 0.6257
Acurácia no teste: 0.6434
```

Além disso, podemos plotar os gráficos de perda e acurácia com os dados de treino e validação para verificar o quão bom foi o treinamento da rede.

```
plt.figure(figsize=(10,4))
plt.subplot(1,2,1)
plt.plot(history.history['loss'], label='treino')
plt.plot(history.history['val_loss'], label='validação')
plt.title('Perda')
plt.legend()

plt.subplot(1,2,2)
plt.plot(history.history['accuracy'], label='treino')
plt.plot(history.history['val_accuracy'], label='validação')
plt.title('Acurácia')
plt.legend()
plt.show()
```



Visualmente podemos ver que as linhas seguem o mesmo caminho e não parece haver qualquer indício de overfitting, apenas a função de perda parece apresentar uma leve distância entre treino e validação, mas nada muito relevante, dado que o treinamento foi interrompido antes que qualquer problema pudesse vir a aparecer.

Ao treinar a rede e obter bons resultados, podemos gravar o modelo em um arquivo `keras` e utilizar ele sempre que precisarmos utilizá-lo sem a necessidade de treinar a rede novamente, pois além de custar tempo, os resultados vão sempre ser relativamente diferentes, não muito distante, pois os parâmetros são os mesmos, mas os resultados vão possuir comportamentos distintos, então para garantir a reprodutibilidade fazemos este procedimento.

```
model.save("higgs_classifier_keras.keras")
print("Modelo gravado como higgs_classifier_keras.keras")
```

```
Modelo gravado como higgs_classifier_keras.keras
```

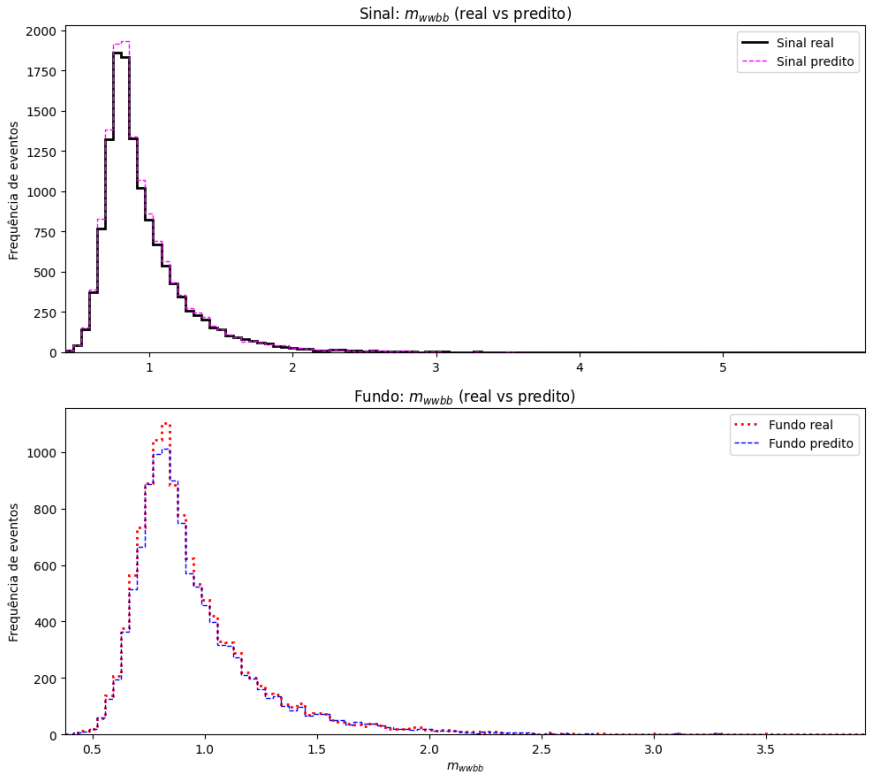
Antes de prosseguir, há uma nota a ser feita em relação à esta última ação. Ao executar o código por completo no Google Colab, a rede neural vai ser treinada novamente e os valores que aparecem em cada `epoch`, a acurácia total final e os gráficos de perda e acurácia, vão ser ligeiramente diferentes, então para evitar que sempre que os testes da rede forem ser executados no Colab tenhamos que re-treinar a rede, utilizaremos o modelo salvo no `higgs_classifier_keras.keras`. Portanto, faremos o download deste modelo, inserimos ele em um drive público, fazemos o download dele dentro do Colab utilizando a biblioteca `gdown` e carregamos o modelo via `load_model`.

```
from tensorflow.keras.models import load_model

gdown.download("https://drive.google.com/uc?id=1QKsAdoLJUO09CdT-6KNYkfLyvcC5yL8o",
↪  "higgs_classifier_keras.keras", quiet=True)

model = load_model("higgs_classifier_keras.keras")
```

Completo o treinamento, o espectro da massa m_{WWbb} pode ser construido para os dados, basta fazer uma simples separação do conjunto `highLevel` em treino e teste para construir esse espectro com as predições de sinal e fundo. Os comandos abaixo são essencialmente para plotar apenas as curvas acima de um histograma (puramente estético e para se parecer mais com o artigo original), então o foco é na predição, onde fazemos com que a probabilidade estimada de que a amostra pertença à classe positiva seja classificada como sinal para valores > 0.5 e como fundo para $\leqslant 0.5$.

```
highLevelTrain, highLevelTest = train_test_split(highLevel, test_size=0.25, random_state=42)
mWWbbTest = highLevelTest['m_wwbb'].values

yPredProb = model.predict(XTest)
yPred = (yPredProb > 0.5).astype(int).flatten()

mWWbb = mWWbbTest

mSinalTrue = mWWbb[yTest == 1]
mSinalPred = mWWbb[yPred == 1]

mFundoTrue = mWWbb[yTest == 0]
mFundoPred = mWWbb[yPred == 0]

def make_bins(data):
    vmin = data.min()
    vmax = data.max()
    return np.linspace(vmin, vmax, 101)

fig, (axSinal, axFundo) = plt.subplots(2, 1, figsize=(10, 9), sharex=False)

binsSinal = make_bins(np.concatenate([mSinalTrue, mSinalPred]))
axSinal.hist(mSinalTrue, bins=binsSinal, histtype='step', linewidth=2, color='black', label='Sinal real')
axSinal.hist(mSinalPred, bins=binsSinal, histtype='step', linewidth=1, color='magenta',
↪  linestyle='dashed', label='Sinal predito')

axSinal.set_title('Sinal: $m_{wwbb}$ (real vs predito)')
axSinal.set_ylabel('Frequência de eventos')
axSinal.set_xlim(binsSinal[0], binsSinal[-1])

legendaSinal = [
```

```
    Line2D([0], [0], color='black', lw=2, label='Sinal real'),
    Line2D([0], [0], color='magenta', lw=1, linestyle='dashed', label='Sinal predito')
]
axSinal.legend(handles=legendaSinal, loc='upper right')

binsFundo = make_bins(np.concatenate([mFundoTrue, mFundoPred]))
axFundo.hist(mFundoTrue, bins=binsFundo, histtype='step', linewidth=2, color='red', linestyle='dotted',
↪  label='Fundo real')
axFundo.hist(mFundoPred, bins=binsFundo, histtype='step', linewidth=1, color='blue', linestyle='dashed',
↪  label='Fundo predito')

axFundo.set_title('Fundo: $m_{wwbb}$ (real vs predito)')
axFundo.set_xlabel('$m_{wwbb}$')
axFundo.set_ylabel('Frequência de eventos')
axFundo.set_xlim(binsFundo[0], binsFundo[-1])

legendaFundo = [
    Line2D([0], [0], color='red', lw=2, linestyle='dotted', label='Fundo real'),
    Line2D([0], [0], color='blue', lw=1, linestyle='dashed', label='Fundo predito')
]
axFundo.legend(handles=legendaFundo, loc='upper right')

plt.tight_layout()
plt.show()
```



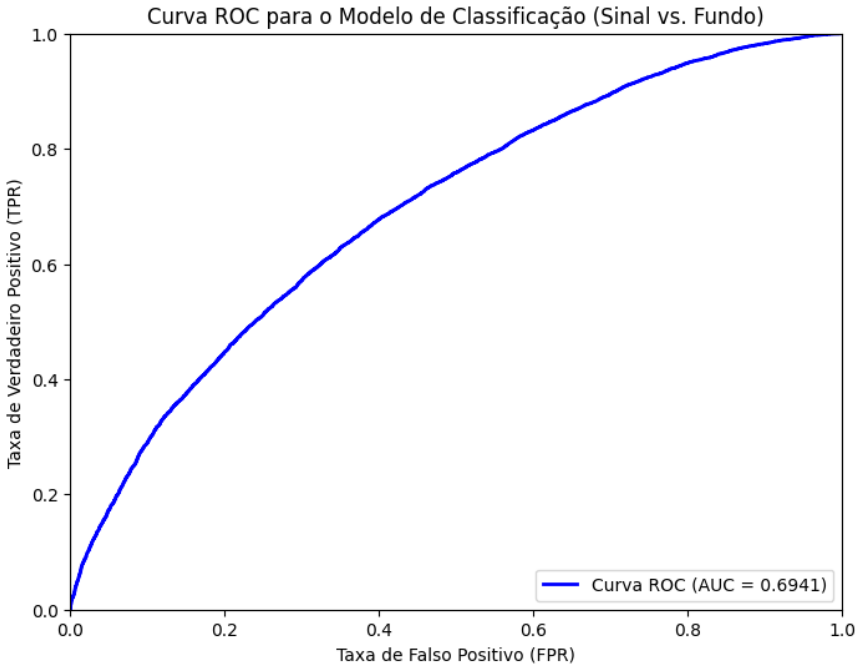
Por fim, para verificar o quão bom são os resultados obtidos a partir do treinamento da rede, podemos utilizar a função `classification_report` do `sklearn` para obter as principais métricas que são a **precisão**, **recall**, **f1-score** e a **acurácia** que já obtivemos no final do treinamento. Essa função vai nos fornecer alguns resultados a mais, mas não daremos atenção à elas, apenas às principais comentadas anteriormente.

```
from sklearn.metrics import classification_report

print(classification_report(yTest, yPred))
```

```
              precision    recall  f1-score   support

         0.0       0.63      0.60      0.61     11792
         1.0       0.65      0.68      0.67     13208

    accuracy                           0.64     25000
   macro avg       0.64      0.64      0.64     25000
weighted avg       0.64      0.64      0.64     25000
```

Obtemos então para o fundo valores de precisão de $\approx 63\%$, um recall de $\approx 60\%$ e um f1-score $\approx 61\%$, o que são valores muito bons levando em conta que usamos uma rede multicamadas de **perceptrons** simples. O mesmo vale pro caso de sinal, onde os valores de precisão ($\approx 65\%$), recall ($\approx 68\%$) e f1-score ($\approx 67\%$) fornecem valores relativamente altos e indicam um bom resultado da predição. Além destas métricas, podemos construir uma curva ROC e determinar a área abaixo desta curva (AUC).

```
from sklearn.metrics import roc_curve, auc
import matplotlib.pyplot as plt

fpr, tpr, thresholds = roc_curve(yTest, yPredProb.flatten())
roc_auc = auc(fpr, tpr)

plt.figure(figsize=(8, 6))
plt.plot(fpr, tpr, color='blue', lw=2, label=f'Curva ROC (AUC = {roc_auc:.4f})')
plt.xlim([0.0, 1.0])
plt.ylim([0.0, 1.0])
plt.xlabel('Taxa de Falso Positivo (FPR)')
plt.ylabel('Taxa de Verdadeiro Positivo (TPR)')
plt.title('Curva ROC para o Modelo de Classificação (Sinal vs. Fundo)')
plt.legend(loc='lower right')
plt.show()
```



O resultado de AUC ≈ 0.69 pode ser considerado muito bom, levando em conta que quanto mais próximo de 1, melhor é o modelo. Além disso, ao olhar para o artigo original, a figura suplementar 1.(a) apresenta um gráfico similar à este, cujo valor de AUC utilizando redes neurais foi de AUC $= 0.73$, concluindo que os resultados obtidos pela rede neural aqui construída são bons o suficiente para extrair a física de interesse.