

Passo 1: Importação de bibliotecas

Importamos bibliotecas essenciais para:

- Manipulação de dados (Pandas, NumPy).
- Processamento de imagens (Pillow).
- Construção e treinamento do modelo de IA (TensorFlow). Configuramos o uso máximo de núcleos disponíveis para otimizar tarefas paralelas.

```
In [1]: # Importar as bibliotecas necessárias
import os
import numpy as np
import pandas as pd
from PIL import Image
import concurrent.futures
from sklearn.model_selection import train_test_split
from sklearn.metrics import accuracy_score
from tensorflow.keras.preprocessing.image import ImageDataGenerator
from tensorflow.keras.applications import DenseNet201
from tensorflow.keras.models import Model, load_model
from tensorflow.keras.layers import Dense, Flatten, Dropout, \
GlobalAveragePooling2D, Input
from tensorflow.keras.optimizers import Adam
from tensorflow.keras.callbacks import ReduceLROnPlateau
from tensorflow.keras.utils import to_categorical

# Configurar o número de workers para multiprocessing
max_workers = os.cpu_count()
```

Passo 2: Configuração dos diretórios

Definimos os caminhos das pastas que contêm as imagens de treino e teste. Esses diretórios são essenciais para organizar e carregar os dados corretamente.

```
In [3]: # Configurar diretórios de treino e teste
train_dir = (r'C:\Users\Danni\Downloads\Nova pasta'
             r'\tcc\Skin cancer ISIC The International '
             r'Skin Imaging Collaboration\Train')
test_dir = (r'C:\Users\Danni\Downloads\Nova pasta'
            r'\tcc\Skin cancer ISIC The International '
            r'Skin Imaging Collaboration\Test')
```

Passo 3: Criação dos DataFrames

Criamos dois DataFrames:

- **train_df** : Contém os caminhos e rótulos das imagens de treino.
- **test_df** : Contém os caminhos e rótulos das imagens de teste. Os rótulos foram atribuídos automaticamente com base nos nomes das subpastas. Por fim, unimos os dois DataFrames em um único conjunto de dados para facilitar a manipulação.

```
In [2]: # Criar DataFrames vazios
train_df = pd.DataFrame(columns=['image_path', 'label'])
test_df = pd.DataFrame(columns=['image_path', 'label'])

# Preencher o DataFrame de treino
for label, directory in enumerate(os.listdir(train_dir)):
    for filename in os.listdir(os.path.join(train_dir, directory)):
        image_path = os.path.join(train_dir, directory, filename)
        train_df = train_df._append({'image_path': image_path, 'label': label}, ignore_index=True)

# Preencher o DataFrame de teste
for label, directory in enumerate(os.listdir(test_dir)):
    for filename in os.listdir(os.path.join(test_dir, directory)):
        image_path = os.path.join(test_dir, directory, filename)
        test_df = test_df._append({'image_path': image_path, 'label': label}, ignore_index=True)

# Concatenar os dois DataFrames
df = pd.concat([train_df, test_df], ignore_index=True)
df.head()
```

Out[2]:

	image_path	label
0	C:\Users\Danni\Downloads\Nova pasta\tcc\Skin c...	0
1	C:\Users\Danni\Downloads\Nova pasta\tcc\Skin c...	0
2	C:\Users\Danni\Downloads\Nova pasta\tcc\Skin c...	0
3	C:\Users\Danni\Downloads\Nova pasta\tcc\Skin c...	0
4	C:\Users\Danni\Downloads\Nova pasta\tcc\Skin c...	0

Passo 4: Divisão dos dados

Separamos o conjunto unificado em:

- **Treino (80%):** Para treinar o modelo.
- **Teste (20%):** Para avaliar o desempenho do modelo. A separação foi feita de maneira aleatória e balanceada usando a função `train_test_split`.

```
In [3]: # Separar caminhos das imagens (X) e rótulos (y)
X = df['image_path'].values
y = df['label'].values

# Dividir os dados em conjuntos de treino e teste
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)
```

Passo 5: Aumento e pré-processamento dos dados

- Configuramos um gerador para aplicar transformações (rotação, zoom, etc.) às imagens, aumentando a diversidade dos dados.
- Redimensionamos todas as imagens para 100x75 pixels e normalizamos os valores dos pixels para o intervalo [0, 1].
- Convertidos os rótulos em `one-hot encoding` para que o modelo reconheça as classes.

```
In [4]: # Definir o gerador de imagens para aumento de dados
datagen = ImageDataGenerator(
    rotation_range=20,
    width_shift_range=0.2,
    height_shift_range=0.2,
    shear_range=0.2,
    zoom_range=0.2,
    horizontal_flip=True,
    fill_mode='nearest'
)

# Função para carregar e redimensionar imagens
def load_image(image_path):
    return np.asarray(Image.open(image_path).resize((100, 75)))

# Carregar imagens usando multiprocessing
with concurrent.futures.ThreadPoolExecutor(max_workers=max_workers) as executor:
    X_train = list(executor.map(load_image, X_train))
    X_test = list(executor.map(load_image, X_test))

# Converter listas em arrays numpy
X_train = np.array(X_train)
X_test = np.array(X_test)

# Normalizar os valores dos pixels
X_train = X_train.astype('float32') / 255.0
X_test = X_test.astype('float32') / 255.0

# Converter os rótulos para one-hot encoding
y_train = to_categorical(y_train)
y_test = to_categorical(y_test)
```

Passo 6: Construção do modelo

Utilizamos o modelo pré-treinado DenseNet201 para:

- Aproveitar suas características previamente aprendidas.

- Adicionar camadas personalizadas específicas para o problema. Congelamos os pesos do modelo base para preservar o aprendizado prévio, otimizando o treinamento.

```
In [5]: # Definir a entrada do modelo
input_shape = (75, 100, 3)
input_tensor = Input(shape=input_shape)

# Carregar o modelo DenseNet201 pré-treinado
base_model = DenseNet201(include_top=False, weights='imagenet', input_tensor=input_tensor, pooling='avg')

# Congelar os pesos do modelo base
base_model.trainable = False

# Adicionar camadas adicionais
x = Dropout(0.5)(base_model.output)
x = Dense(512, activation='relu')(x)
output = Dense(9, activation='softmax')(x)

# Criar o modelo final
model = Model(inputs=input_tensor, outputs=output)

# Compilar o modelo
model.compile(optimizer=Adam(learning_rate=0.001), loss='categorical_crossentropy', metrics=['accuracy'])
```

Passo 7: Treinamento do modelo

Treinamos o modelo com:

- **Gerador de dados** (datagen) para incluir as transformações das imagens.
- **Callback** `ReduceLROnPlateau` para ajustar dinamicamente a taxa de aprendizado quando o desempenho estabiliza. O treinamento foi realizado durante 80 épocas, analisando a perda e a acurácia ao longo do tempo.

Aviso de Warning após treinamento do modelo

- **WARNING:** Nós não estamos usando diretamente PyDataset ou passando argumentos incorretos para fit(). Ele é um aviso genérico emitido devido à forma como Keras/TensorFlow lidam internamente com datasets, podendo ser ignorado.

```
In [6]: # Treinar o modelo
history = model.fit(
    datagen.flow(X_train, y_train, batch_size=32),
    epochs=80,
    validation_data=(X_test, y_test),
    callbacks=[ReduceLROnPlateau(factor=0.5, patience=3, min_lr=0.00001)]
)
```

```
C:\Users\Danni\AppData\Local\Programs\Python\Python311\Lib\site-packages\keras\src\trainers\data_adapters\py_dataset_adapter.py:121: UserWarning: Your `PyDataset` class should call `super().__init__(**kwargs)` in its constructor. `**kwargs` can include `workers`, `use_multiprocessing`, `max_queue_size`. Do not pass these arguments to `fit()`, as they will be ignored.
  self._warn_if_super_not_called()
```

Epoch 1/80
59/59  111s 1s/step - accuracy: 0.2826 - loss: 2.2732 - val_accuracy: 0.4661 - val_loss: 1.5701 - learning_rate: 0.0010

Epoch 2/80
59/59  51s 871ms/step - accuracy: 0.3922 - loss: 1.6995 - val_accuracy: 0.4597 - val_loss: 1.5100 - learning_rate: 0.0010

Epoch 3/80
59/59  50s 855ms/step - accuracy: 0.4689 - loss: 1.5308 - val_accuracy: 0.5021 - val_loss: 1.3922 - learning_rate: 0.0010

Epoch 4/80
59/59  54s 914ms/step - accuracy: 0.4549 - loss: 1.5088 - val_accuracy: 0.5445 - val_loss: 1.3471 - learning_rate: 0.0010


Epoch 5/80
59/59  51s 870ms/step - accuracy: 0.4667 - loss: 1.4356 - val_accuracy: 0.5636 - val_loss: 1.2931 - learning_rate: 0.0010

Epoch 6/80
59/59  49s 834ms/step - accuracy: 0.4752 - loss: 1.4334 - val_accuracy: 0.5445 - val_loss: 1.2977 - learning_rate: 0.0010

Epoch 7/80
59/59  51s 858ms/step - accuracy: 0.5114 - loss: 1.3964 - val_accuracy: 0.5551 - val_loss: 1.2696 - learning_rate: 0.0010

Epoch 8/80
59/59  49s 827ms/step - accuracy: 0.5145 - loss: 1.3279 - val_accuracy: 0.5678 - val_loss: 1.2853 - learning_rate: 0.0010

Epoch 9/80
59/59  47s 803ms/step - accuracy: 0.5229 - loss: 1.3000 - val_accuracy: 0.5487 - val_loss: 1.2527 - learning_rate: 0.0010

Epoch 10/80
59/59  47s 805ms/step - accuracy: 0.5167 - loss: 1.3374 - val_accuracy: 0.5699 - val_loss: 1.2216 - learning_rate: 0.0010


Epoch 11/80
59/59  47s 794ms/step - accuracy: 0.5132 - loss: 1.3169 - val_accuracy: 0.5551 - val_loss: 1.2164 - learning_rate: 0.0010

Epoch 12/80
59/59  50s 844ms/step - accuracy: 0.5446 - loss: 1.2582 - val_accuracy: 0.5657 - val_loss: 1.2473 - learning_rate: 0.0010


Epoch 13/80
59/59  47s 789ms/step - accuracy: 0.5417 - loss: 1.2853 - val_accuracy: 0.5678 - val_loss: 1.2324 - learning_rate: 0.0010

Epoch 14/80
59/59  47s 794ms/step - accuracy: 0.5425 - loss: 1.2786 - val_accuracy: 0.5614 - val_loss: 1.2489 - learning_rate: 0.0010

g_rate: 0.0010
Epoch 15/80
59/59 ————— 46s 784ms/step - accuracy: 0.5573 - loss: 1.2181 - val_accuracy: 0.5763 - val_loss: 1.2003 - learnin
g_rate: 5.0000e-04
Epoch 16/80
59/59 ————— 48s 808ms/step - accuracy: 0.5474 - loss: 1.2205 - val_accuracy: 0.5636 - val_loss: 1.1947 - learnin
g_rate: 5.0000e-04
Epoch 17/80
59/59 ————— 51s 858ms/step - accuracy: 0.5638 - loss: 1.1861 - val_accuracy: 0.5593 - val_loss: 1.1914 - learnin
g_rate: 5.0000e-04
Epoch 18/80
59/59 ————— 54s 923ms/step - accuracy: 0.5640 - loss: 1.2415 - val_accuracy: 0.5593 - val_loss: 1.1873 - learnin
g_rate: 5.0000e-04
Epoch 19/80
59/59 ————— 51s 860ms/step - accuracy: 0.5518 - loss: 1.2266 - val_accuracy: 0.5805 - val_loss: 1.1652 - learnin
g_rate: 5.0000e-04
Epoch 20/80
59/59 ————— 46s 787ms/step - accuracy: 0.5727 - loss: 1.2021 - val_accuracy: 0.5784 - val_loss: 1.1758 - learnin
g_rate: 5.0000e-04
Epoch 21/80
59/59 ————— 48s 806ms/step - accuracy: 0.5523 - loss: 1.2024 - val_accuracy: 0.5932 - val_loss: 1.1741 - learnin
g_rate: 5.0000e-04
Epoch 22/80
59/59 ————— 46s 785ms/step - accuracy: 0.5624 - loss: 1.1966 - val_accuracy: 0.5742 - val_loss: 1.1919 - learnin
g_rate: 5.0000e-04
Epoch 23/80
59/59 ————— 47s 793ms/step - accuracy: 0.5848 - loss: 1.1343 - val_accuracy: 0.5763 - val_loss: 1.1559 - learnin
g_rate: 2.5000e-04
Epoch 24/80
59/59 ————— 47s 804ms/step - accuracy: 0.5837 - loss: 1.1173 - val_accuracy: 0.5890 - val_loss: 1.1472 - learnin
g_rate: 2.5000e-04
Epoch 25/80
59/59 ————— 46s 781ms/step - accuracy: 0.5612 - loss: 1.2189 - val_accuracy: 0.5742 - val_loss: 1.1732 - learnin
g_rate: 2.5000e-04
Epoch 26/80
59/59 ————— 47s 789ms/step - accuracy: 0.5852 - loss: 1.1382 - val_accuracy: 0.5784 - val_loss: 1.1731 - learnin
g_rate: 2.5000e-04
Epoch 27/80
59/59 ————— 47s 799ms/step - accuracy: 0.5746 - loss: 1.1357 - val_accuracy: 0.5742 - val_loss: 1.1550 - learnin
g_rate: 2.5000e-04
Epoch 28/80


59/59  47s 790ms/step - accuracy: 0.5725 - loss: 1.1115 - val_accuracy: 0.5699 - val_loss: 1.1694 - learning_rate: 1.2500e-04
Epoch 29/80

59/59  46s 787ms/step - accuracy: 0.6050 - loss: 1.1112 - val_accuracy: 0.5720 - val_loss: 1.1712 - learning_rate: 1.2500e-04
Epoch 30/80


59/59  47s 797ms/step - accuracy: 0.5707 - loss: 1.1495 - val_accuracy: 0.5826 - val_loss: 1.1605 - learning_rate: 1.2500e-04
Epoch 31/80


59/59  47s 789ms/step - accuracy: 0.6171 - loss: 1.0537 - val_accuracy: 0.5805 - val_loss: 1.1593 - learning_rate: 6.2500e-05
Epoch 32/80

59/59  47s 791ms/step - accuracy: 0.6057 - loss: 1.0714 - val_accuracy: 0.5784 - val_loss: 1.1604 - learning_rate: 6.2500e-05
Epoch 33/80


59/59  47s 795ms/step - accuracy: 0.6064 - loss: 1.0770 - val_accuracy: 0.5847 - val_loss: 1.1539 - learning_rate: 6.2500e-05
Epoch 34/80

59/59  47s 800ms/step - accuracy: 0.6120 - loss: 1.0948 - val_accuracy: 0.5826 - val_loss: 1.1519 - learning_rate: 3.1250e-05
Epoch 35/80


59/59  47s 803ms/step - accuracy: 0.6063 - loss: 1.0882 - val_accuracy: 0.5805 - val_loss: 1.1465 - learning_rate: 3.1250e-05
Epoch 36/80


59/59  47s 800ms/step - accuracy: 0.5995 - loss: 1.0921 - val_accuracy: 0.5784 - val_loss: 1.1470 - learning_rate: 3.1250e-05
Epoch 37/80


59/59  47s 802ms/step - accuracy: 0.6034 - loss: 1.1024 - val_accuracy: 0.5763 - val_loss: 1.1530 - learning_rate: 3.1250e-05
Epoch 38/80


59/59  47s 798ms/step - accuracy: 0.6010 - loss: 1.1070 - val_accuracy: 0.5742 - val_loss: 1.1546 - learning_rate: 3.1250e-05
Epoch 39/80


59/59  46s 788ms/step - accuracy: 0.5859 - loss: 1.1512 - val_accuracy: 0.5742 - val_loss: 1.1519 - learning_rate: 1.5625e-05
Epoch 40/80


59/59  47s 798ms/step - accuracy: 0.5992 - loss: 1.1040 - val_accuracy: 0.5720 - val_loss: 1.1532 - learning_rate: 1.5625e-05
Epoch 41/80


59/59  47s 798ms/step - accuracy: 0.6092 - loss: 1.0979 - val_accuracy: 0.5763 - val_loss: 1.1524 - learning_rate: 1.5625e-05


Epoch 42/80
59/59  47s 797ms/step - accuracy: 0.6347 - loss: 1.0499 - val_accuracy: 0.5742 - val_loss: 1.1513 - learning_rate: 1.0000e-05


Epoch 43/80
59/59  46s 784ms/step - accuracy: 0.6010 - loss: 1.1158 - val_accuracy: 0.5742 - val_loss: 1.1509 - learning_rate: 1.0000e-05


Epoch 44/80
59/59  48s 811ms/step - accuracy: 0.5841 - loss: 1.1118 - val_accuracy: 0.5742 - val_loss: 1.1519 - learning_rate: 1.0000e-05


Epoch 45/80
59/59  47s 789ms/step - accuracy: 0.6030 - loss: 1.0640 - val_accuracy: 0.5720 - val_loss: 1.1518 - learning_rate: 1.0000e-05


Epoch 46/80
59/59  47s 803ms/step - accuracy: 0.6010 - loss: 1.0887 - val_accuracy: 0.5699 - val_loss: 1.1525 - learning_rate: 1.0000e-05


Epoch 47/80
59/59  46s 786ms/step - accuracy: 0.5944 - loss: 1.1095 - val_accuracy: 0.5699 - val_loss: 1.1538 - learning_rate: 1.0000e-05


Epoch 48/80
59/59  39s 652ms/step - accuracy: 0.5815 - loss: 1.1169 - val_accuracy: 0.5720 - val_loss: 1.1526 - learning_rate: 1.0000e-05


Epoch 49/80
59/59  34s 578ms/step - accuracy: 0.5952 - loss: 1.1136 - val_accuracy: 0.5742 - val_loss: 1.1532 - learning_rate: 1.0000e-05


Epoch 50/80
59/59  34s 580ms/step - accuracy: 0.5916 - loss: 1.0922 - val_accuracy: 0.5742 - val_loss: 1.1536 - learning_rate: 1.0000e-05

Epoch 51/80
59/59  36s 604ms/step - accuracy: 0.5815 - loss: 1.1174 - val_accuracy: 0.5742 - val_loss: 1.1527 - learning_rate: 1.0000e-05

Epoch 52/80
59/59  37s 629ms/step - accuracy: 0.6084 - loss: 1.0463 - val_accuracy: 0.5720 - val_loss: 1.1507 - learning_rate: 1.0000e-05

Epoch 53/80
59/59  36s 609ms/step - accuracy: 0.6015 - loss: 1.0938 - val_accuracy: 0.5720 - val_loss: 1.1514 - learning_rate: 1.0000e-05

Epoch 54/80
59/59  35s 588ms/step - accuracy: 0.5977 - loss: 1.1157 - val_accuracy: 0.5742 - val_loss: 1.1503 - learning_rate: 1.0000e-05

Epoch 55/80
59/59  35s 598ms/step - accuracy: 0.5958 - loss: 1.1033 - val_accuracy: 0.5742 - val_loss: 1.1497 - learning_rate: 1.0000e-05

```
g_rate: 1.0000e-05
Epoch 56/80
59/59 ————— 35s 597ms/step - accuracy: 0.6003 - loss: 1.0984 - val_accuracy: 0.5742 - val_loss: 1.1498 - learnin
g_rate: 1.0000e-05
Epoch 57/80
59/59 ————— 36s 612ms/step - accuracy: 0.5816 - loss: 1.1245 - val_accuracy: 0.5763 - val_loss: 1.1506 - learnin
g_rate: 1.0000e-05
Epoch 58/80
59/59 ————— 34s 580ms/step - accuracy: 0.6239 - loss: 1.0568 - val_accuracy: 0.5763 - val_loss: 1.1532 - learnin
g_rate: 1.0000e-05
Epoch 59/80
59/59 ————— 36s 617ms/step - accuracy: 0.6059 - loss: 1.1011 - val_accuracy: 0.5763 - val_loss: 1.1509 - learnin
g_rate: 1.0000e-05
Epoch 60/80
59/59 ————— 38s 654ms/step - accuracy: 0.6079 - loss: 1.0948 - val_accuracy: 0.5784 - val_loss: 1.1516 - learnin
g_rate: 1.0000e-05
Epoch 61/80
59/59 ————— 35s 585ms/step - accuracy: 0.6030 - loss: 1.1128 - val_accuracy: 0.5784 - val_loss: 1.1516 - learnin
g_rate: 1.0000e-05
Epoch 62/80
59/59 ————— 35s 594ms/step - accuracy: 0.6159 - loss: 1.0507 - val_accuracy: 0.5784 - val_loss: 1.1504 - learnin
g_rate: 1.0000e-05
Epoch 63/80
59/59 ————— 35s 594ms/step - accuracy: 0.6192 - loss: 1.0694 - val_accuracy: 0.5784 - val_loss: 1.1500 - learnin
g_rate: 1.0000e-05
Epoch 64/80
59/59 ————— 35s 586ms/step - accuracy: 0.5944 - loss: 1.1060 - val_accuracy: 0.5763 - val_loss: 1.1509 - learnin
g_rate: 1.0000e-05
Epoch 65/80
59/59 ————— 34s 581ms/step - accuracy: 0.6123 - loss: 1.0541 - val_accuracy: 0.5784 - val_loss: 1.1496 - learnin
g_rate: 1.0000e-05
Epoch 66/80
59/59 ————— 36s 609ms/step - accuracy: 0.5970 - loss: 1.0963 - val_accuracy: 0.5805 - val_loss: 1.1482 - learnin
g_rate: 1.0000e-05
Epoch 67/80
59/59 ————— 36s 613ms/step - accuracy: 0.5949 - loss: 1.1290 - val_accuracy: 0.5805 - val_loss: 1.1473 - learnin
g_rate: 1.0000e-05
Epoch 68/80
59/59 ————— 35s 589ms/step - accuracy: 0.5939 - loss: 1.1111 - val_accuracy: 0.5784 - val_loss: 1.1485 - learnin
g_rate: 1.0000e-05
Epoch 69/80
```

59/59 ————— 45s 773ms/step - accuracy: 0.5808 - loss: 1.1248 - val_accuracy: 0.5763 - val_loss: 1.1499 - learning_rate: 1.0000e-05
Epoch 70/80

59/59 ————— 50s 835ms/step - accuracy: 0.5976 - loss: 1.1193 - val_accuracy: 0.5784 - val_loss: 1.1489 - learning_rate: 1.0000e-05
Epoch 71/80

59/59 ————— 50s 836ms/step - accuracy: 0.6190 - loss: 1.0800 - val_accuracy: 0.5742 - val_loss: 1.1503 - learning_rate: 1.0000e-05
Epoch 72/80

59/59 ————— 48s 821ms/step - accuracy: 0.6166 - loss: 1.0495 - val_accuracy: 0.5742 - val_loss: 1.1493 - learning_rate: 1.0000e-05
Epoch 73/80

59/59 ————— 53s 896ms/step - accuracy: 0.5732 - loss: 1.1461 - val_accuracy: 0.5784 - val_loss: 1.1486 - learning_rate: 1.0000e-05
Epoch 74/80

59/59 ————— 52s 878ms/step - accuracy: 0.6063 - loss: 1.0604 - val_accuracy: 0.5742 - val_loss: 1.1497 - learning_rate: 1.0000e-05
Epoch 75/80

59/59 ————— 53s 899ms/step - accuracy: 0.5906 - loss: 1.1298 - val_accuracy: 0.5763 - val_loss: 1.1488 - learning_rate: 1.0000e-05
Epoch 76/80

59/59 ————— 51s 869ms/step - accuracy: 0.6032 - loss: 1.0823 - val_accuracy: 0.5784 - val_loss: 1.1472 - learning_rate: 1.0000e-05
Epoch 77/80

59/59 ————— 51s 861ms/step - accuracy: 0.6101 - loss: 1.0664 - val_accuracy: 0.5763 - val_loss: 1.1480 - learning_rate: 1.0000e-05
Epoch 78/80

59/59 ————— 50s 834ms/step - accuracy: 0.6145 - loss: 1.0897 - val_accuracy: 0.5742 - val_loss: 1.1489 - learning_rate: 1.0000e-05
Epoch 79/80

59/59 ————— 49s 834ms/step - accuracy: 0.6205 - loss: 1.0575 - val_accuracy: 0.5784 - val_loss: 1.1481 - learning_rate: 1.0000e-05
Epoch 80/80

59/59 ————— 53s 888ms/step - accuracy: 0.6186 - loss: 1.0453 - val_accuracy: 0.5805 - val_loss: 1.1480 - learning_rate: 1.0000e-05

Gráficos de Perda e Acurácia do Modelo

O que são?

Os gráficos de perda e acurácia mostram como o modelo se comportou durante o treinamento e a validação ao longo das épocas.

Gráfico de Perda

- **O que representa?**
 - A **perda** é uma medida do erro do modelo ao prever os dados.
 - Quanto menor a perda, melhor o desempenho do modelo.
 - **Análise:**
 - A perda diminui ao longo das épocas para o conjunto de treino e validação.
 - Se houver uma diferença muito grande entre os dois conjuntos, pode indicar **overfitting** (o modelo está memorizando os dados de treino, ao invés de conhecer as características gerais que deveriam ser reconhecidas)
-

Gráfico de Acurácia

- **O que representa?**
 - A **acurácia** mede a proporção de previsões corretas do modelo.
 - Valores mais altos indicam que o modelo está acertando mais.
 - **Análise:**
 - A acurácia geralmente aumenta ao longo das épocas para o conjunto de treino.
 - A estabilização ou queda na acurácia de validação pode indicar **underfitting** ou **overfitting**.
-

Como interpretar?

- **Convergência:**

- A perda deve diminuir e a acurácia deve aumentar ao longo das épocas.
- Diferenças entre treino e validação devem ser mínimas para indicar um bom generalizador.

- **Melhor desempenho:**

- Um modelo equilibrado apresenta acurácia alta e perda baixa tanto para treino quanto validação.
-

Como foi gerado?

Os gráficos foram criados a partir do histórico (`history`) do treinamento:

- Perda: `loss` e `val_loss` .
- Acurácia: `accuracy` e `val_accuracy` .

Esses valores foram extraídos e visualizados utilizando a biblioteca `Matplotlib` .

Com esses gráficos, conseguimos identificar o comportamento do modelo e avaliar sua capacidade de generalização.

```
In [7]: # Obter os dados do histórico
import matplotlib.pyplot as plt

train_loss = history.history['loss']
val_loss = history.history['val_loss']
train_accuracy = history.history['accuracy']
val_accuracy = history.history['val_accuracy']

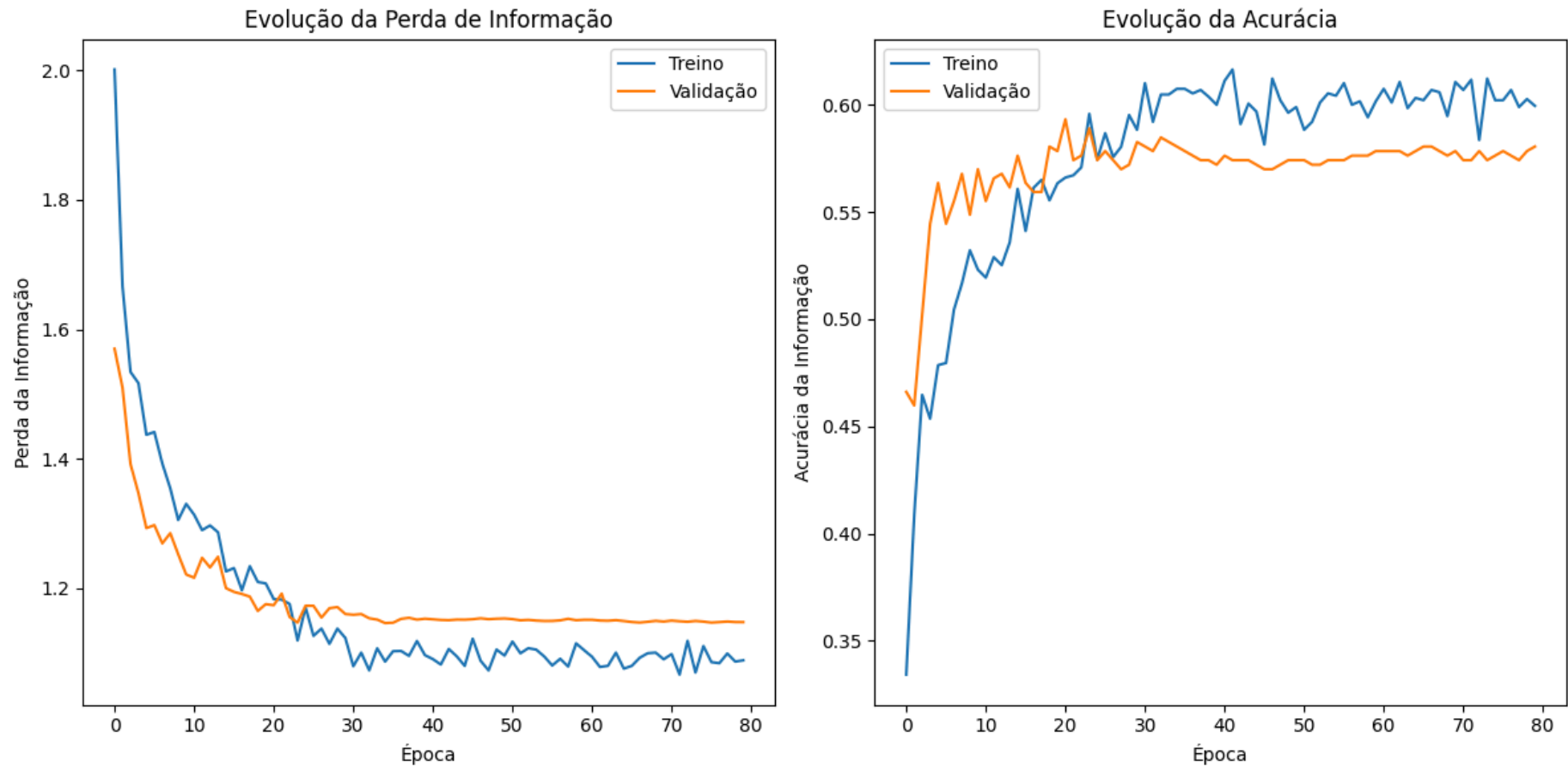
# Plotar a perda
plt.figure(figsize=(12, 6))
plt.subplot(1, 2, 1)
plt.plot(train_loss, label='Treino')
plt.plot(val_loss, label='Validação')
plt.title('Evolução da Perda de Informação')
plt.xlabel('Época')
plt.ylabel('Perda da Informação')
```

```
plt.legend()

# Plotar a acurácia
plt.subplot(1, 2, 2)
plt.plot(train_accuracy, label='Treino')
plt.plot(val_accuracy, label='Validação')
plt.title('Evolução da Acurácia')
plt.xlabel('Época')
plt.ylabel('Acurácia da Informação')
plt.legend()

plt.tight_layout()
plt.show()# Obter os dados do histórico
train_loss = history.history['loss']
val_loss = history.history['val_loss']
train_accuracy = history.history['accuracy']
val_accuracy = history.history['val_accuracy']

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



<Figure size 640x480 with 0 Axes>

Passo 8: Avaliação do modelo

- Avaliamos o modelo nos dados de teste, calculando sua acurácia.
- Salvamos o modelo treinado no formato `.keras` para reutilização futura.

```
In [8]: # Avaliar o modelo
loss, accuracy = model.evaluate(X_test, y_test, verbose=0)
print(f"Acurácia nos dados de teste: {accuracy * 100:.2f}%")
```



```
# Salvar o modelo treinado
model.save('skin_cancer_model_with_other.keras')
```

Acurácia nos dados de teste: 58.05%

Passo 9: Predição com o modelo treinado

Carregamos o modelo salvo e:

- Processamos uma nova imagem, redimensionando-a e normalizando os valores dos pixels.
- Fizemos a predição para identificar a classe da lesão de pele.

```
In [9]: # Carregar o modelo salvo
loaded_model = load_model('skin_cancer_model_with_other.keras', compile=False)

# Função para carregar uma nova imagem
def load_single_image(image_path):
    img = Image.open(image_path).resize((100, 75))
    img_array = np.asarray(img) / 255.0
    return np.expand_dims(img_array, axis=0)

# Caminho para uma nova imagem
new_image_path = r"C:\Users\Danni\Downloads\Nova pasta\tcc\IMAGEM_TESTE.jpg"
new_image = load_single_image(new_image_path)

# Fazer previsão
predictions = loaded_model.predict(new_image)
classes = ['Melanoma', 'Nevus', 'Benign Keratosis',
           'Basal Cell Carcinoma', 'Squamous Cell Carcinoma', 'Vascular Lesion',
           'Actinic Keratosis', 'Dermatofibroma', 'Seborrheic Keratosis', 'Unknown']

predicted_class = classes[np.argmax(predictions)]
print(f"A doença prevista é: {predicted_class}")
```

1/1 ————— 8s 8s/step

A doença prevista é: Benign Keratosis

Matriz de Confusão

O que é?

A matriz de confusão é uma tabela que mostra:

- **Predições corretas:** Representadas na diagonal principal.
- **Erros de classificação:** Representados fora da diagonal principal.

Ela ajuda a entender:

- Quais classes o modelo está acertando mais.
 - Quais classes estão sendo mais confundidas.
-

Como foi criada?

1. **Predições:** Usamos o conjunto de teste (`X_test`) para gerar as classes previstas.
 2. **Rótulos Verdadeiros:** Extraímos as classes verdadeiras do conjunto de teste (`y_test`).
 3. **Matriz de Confusão:** Utilizamos a função `confusion_matrix` da biblioteca `sklearn`.
 4. **Visualização:** Utilizamos o `ConfusionMatrixDisplay` para exibir a matriz com os nomes das classes e um gradiente de cores.
-

Análise dos Resultados

- **Diagonal principal:** Classes que o modelo classificou corretamente.
- **Valores altos fora da diagonal:** Indicam as classes que o modelo tem dificuldade em distinguir.

Essa análise ajuda a identificar áreas em que o modelo pode ser aprimorado, como:

- Aplicar mais técnicas de aumento de dados para classes específicas.

- Ajustar a arquitetura do modelo para melhorar a classificação.

```
In [12]: from sklearn.metrics import confusion_matrix, ConfusionMatrixDisplay

# Fazer predições no conjunto de teste
y_pred = model.predict(X_test)
y_pred_classes = np.argmax(y_pred, axis=1) # Converter previsões em classes
y_true_classes = np.argmax(y_test, axis=1) # Rótulos verdadeiros

# Ajustar os rótulos das classes (remover 'Unknown')
classes = ['Melanoma', 'Nevus', 'Queratoses Benignas',
           'Carcinoma Basocelular', 'Carcinoma Espinocelular',
           'Lesão Vascular', 'Queratoses Actínicas',
           'Dermatofibroma', 'Queratoses Seborreicas']

# Criar matriz de confusão
cm = confusion_matrix(y_true_classes, y_pred_classes)

# Exibir matriz de confusão com o mapa de cores ajustado
disp = ConfusionMatrixDisplay(confusion_matrix=cm, display_labels=classes)
disp.plot(cmap="Blues", xticks_rotation="vertical")
plt.title("Matriz de Confusão - Resultados do Modelo")
plt.xlabel("Classe Prevista")
plt.ylabel("Classe Verdadeira")
plt.show()
```

15/15 ————— 11s 711ms/step

