AULA 7
GERAÇÃO DE DADOS

1. Objetivos

- Apresentar o princípio básico de funcionamento dos geradores de dados.
- Exemplo de um gerador de imagens simples.
- Apresentar o gerador de imagens do Keras.
- Apresentar como treinar uma RNA usando o gerador de imagens do Keras.
- Apresentar um exemplo de como criar um gerador de imagens com algumas funções não disponíveis no Keras.
- Apresentar a camada `lambda` para realizar cálculos simples dentro de uma RNA.

2. Introdução

- Um problema comum na prática real de "deep learning" é ter um conjunto de dados muito grande que não cabe na memória do computador.
- Na medida em que a área de aprendizado de máquina avança, esse problema se torna cada vez mais comum. Atualmente esse já é um dos desafios no campo de visão, onde grandes conjuntos de dados de imagens e arquivos de vídeo são processados.
- Em alguns casos, mesmo a configuração de computador mais avançada não tem espaço de memória suficiente para processar os dados da maneira como fizemos até agora $\Rightarrow$ é necessária outra forma de lidar com grandes conjuntos de dados.
- **Como fazer para treinar uma RNA com um conjunto de dados que não cabe na memória do computador?**

  Solução é carregar os dados durante o treinamento lote por lote e a após cada lote ser usado ele é descartado.
- Outro problema comum quando se processa imagens é a pequena variedade das imagens que estão disponíveis para serem usadas. Não é possível obter exemplos de imagens com todas as variedades possíveis, pois o número de variações de objetos, pontos de vista e outras variações é infinito.

- Além desses dois aspectos, as imagens utilizadas nos trabalhos realizados até o momento já estavam preparadas, ou seja, todas tinham a mesma dimensão, os objetos de interesse estavam centrados, todas as imagens tinham a mesma proporção de tela, as imagens tinham dimensão pequena etc ⇒ na prática real nunca se tem imagens "comportadas" dessa forma.

- Na prática real as imagens não estão preparadas para serem usadas por uma RNA ⇒ assim, as RNAs têm que ser capazes de lidar com imagens de qualquer tipo, ou seja, tamanhos diferentes, objetos de interesse não centrados, diferentes proporções de tela, diferentes luminosidades etc.

- Um exemplo real de imagens disponíveis para treinamento é apresentado na Figura 1 para o problema original de identificar se uma imagem mostra ou não um gato.

  Observa-se que as imagens tem proporções de tela diferentes, às vezes possuem mais de um gato, os gatos não estão centrados ect.

Figura 1. Imagens reais de gatos usado no problema original de identificar se uma imagem mostra ou não gato.

- A solução para esses dois problemas (conjunto de dados muito grande e imagens pouco variadas) é usar um gerador de dados. Um gerador de dados carrega as imagens lote a lote, em tempo real durante o treinamento da RNA, e processa as imagens de forma a colocá-las todas na forma necessária, ou seja, mesma dimensão, normalizadas, mesma proporção de tela etc. Além disso, um gerador de dados é capaz de realizar transformações nas imagens de forma a alterá-las em relação às imagens originais, criando algumas variações.

3. Princípio básico de funcionamento de um gerador de dados

- Um gerador de dados é inerente da linguagem Python, sendo que gera valores, um de cada vez, a partir de uma determinada sequência, em vez de fornecer a totalidade da sequência de uma só vez.

- Função regular versus gerador de dados:

```
# Função regular
def function_a():
    return "a" #

# Gerador
def generator_a():
    yield "a"
```

  - Em um gerador de dados o comando `yield` substitui o `return`.

- Ao chamar uma função regular, o código da função é executado e o resultado é retornado.

```
# Executa function_a
function_a()
```

```
'a'
```

- Para que um gerador retorne seus valores, é necessário passar o gerador para a função `next()`

```
# Pergunta ao gerador qual o próximo item
next(generator_a())
```

```
'a'
```

  - Nesse exemplo o gerador só tem um valor para retornar ⇒ não é um gerador é útil.

- Para um gerador retornar uma sequência de números, sendo um a cada vez que é chamado, podemos incluir vários comandos `yield` ⇒ esses comandos formam a sequência que o gerador produz

- No código abaixo é definido um gerador de nome `multi_gerador` e o mesmo é associado à variável `mg`.

```
def multi_gerador():
    yield "a"
    yield "b"
    yield "c"

mg = multi_gerador()
```

- Se usarmos esse gerador, passando `mg` para `next()`, obtemos o próximo valor até chegar ao final da lista.

```
print(next(mg))
print(next(mg))
print(next(mg))
print(next(mg))
```

```
a
b
c
---------------------------------------------------------------------------
StopIteration                             Traceback (most recent call last)
<ipython-input-5-4ccf14a1b0d8> in <module>
      2 print(next(mg))
      3 print(next(mg))
----> 4 print(next(mg))
StopIteration:
```

- Observe que se tentarmos usar o gerador após atingir o final da lista, uma mensagem de erro é gerada.

4. Gerador de imagens simples

- Um gerador de imagens pode, por exemplo, carregar imagens de um diretório e as transformar de acordo com o código definido no gerador.

- Para usar um gerador de imagens para realizar essa tarefa, primeiramente precisamos indicar ao gerador o diretório onde estão as imagens e gerar a lista dos arquivos que queremos usar.

- No código abaixo é definido o diretório onde estão as imagens (`Face_Imagens`) e salvo na variável `face_path`. Além disso, é criada uma lista com os nomes dos arquivos tipo `*.jpg`, que serão usadas no gerador, na variável `face_img_paths`.

```
# Importa bilbliotecas glob e os
from glob import glob
import os

# Define diretório onde se encontram as imagens
face_path = 'Face_Imagens'

# Escolhe tipos de arquivos desejados
glob_imgs = os.path.join(face_path, '*.jpg')

# Cria lista dos nomes dos arquivos
face_img_paths = glob(glob_imgs)

# Imprime nomes e paths dos 5 primeiros arquivos da lista
print(face_img_paths[:5])
```

```
['Face_Imagens\\4000.jpg', 'Face_Imagens\\4001.jpg',
'Face_Imagens\\4002.jpg', 'Face_Imagens\\4003.jpg',
'Face_Imagens\\4004.jpg', 'Face_Imagens\\4005.jpg',
'Face_Imagens\\4006.jpg', 'Face_Imagens\\4007.jpg',
'Face_Imagens\\4008.jpg', 'Face_Imagens\\4009.jpg']
```

- A biblioteca `glob` do Python permite listar arquivos de um diretório que possuem terminações iguais, na sequência em que se encontram. No código listamos os arquivos de imagens do tipo `*.jpg`. Para mais detalhes sobre a biblioteca `glob` ver https://docs.python.org/2/library/glob.html.

- A biblioteca `os` do Python fornece uma maneira de usar funções do sistema operacional do computador, seja ele Windows, Mac ou Linux. Para mais detalhes sobre a biblioteca `os` ver https://docs.python.org/3/library/os.html.

- Existem muitas funções úteis na biblioteca `os`, uma delas é a função `listdir(diretório)`, que serve para criar uma lista de todos os arquivo (não importando o tipo) dentro de um diretório.

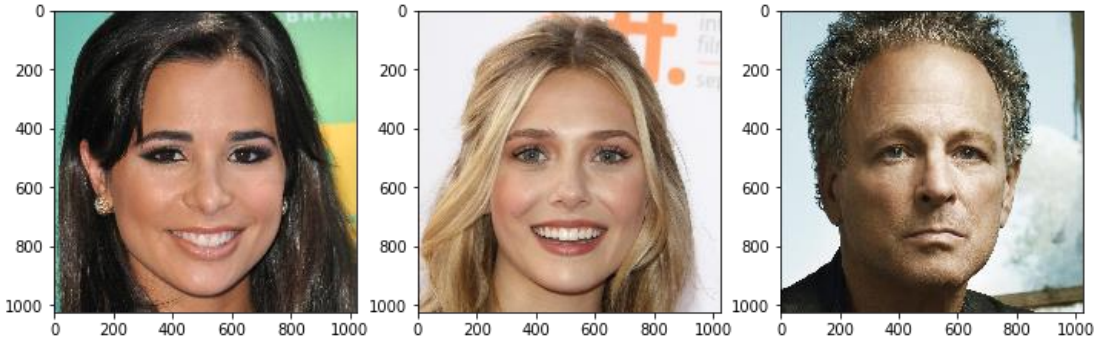
➢ No código a seguir é criado um gerador de imagens simples. Nesse gerador iteramos sobre os nomes dos arquivos que queremos usar e abrimos esses arquivos usando a função `imread` da biblioteca `skimage`.

```
# Importa função imread da biblioteca skyimage.io
from skimage.io import imread

# Cria gerador de imagem simples
def simple_image_gen(img_paths):
    for img_path in img_paths:
        # Carrega a imagem
        img = imread(img_path)

        # Yield imagem
        yield img
```

- A biblioteca scikit-image (também conhecida como `skimage`) fornece uma coleção de funções para processamento de imagem e visão computacional, sendo uma delas a função `imread` para carregar uma imagem. Para mais detalhes sobre essa biblioteca ver https://scikit-image.org/.

➢ Para usar o gerador de imagens simples (`simple_image_gen`), primeiramente tem-se que associá-lo com a lista de nomes e caminhos das imagens desejadas, para depois poder usá-lo para carregar as imagens com o comando `next()`, conforme mostrado no código a seguir.

```
# Inicializa o gerador
img = simple_image_gen(face_img_paths)

# Usa o gerador 3 vezes
first_img = next(img)
sec_img = next(img)
trd_img = next(img)

# Mostra as imagens geradas
f, pos = plt.subplots(1, 3, figsize=(12, 12))
pos[0].imshow(first_img)
pos[1].imshow(sec_img)
pos[2].imshow(trd_img)
plt.show()
```



- Observe que a dimensão das imagens é 1024x1024 pixels ⇒ essa dimensão pode ser alterada dentro do gerador com será visto adiante.

5. Gerador de imagens do TensorFlow-Keras

- O TensorFlow-Keras fornece classes específicas para criar e configurar geradores de dados para carregar e processar imagens.

- O Keras possui três métodos para treinamento, avaliação e previsão de RNAs que usam geradores de dados: `fit_generator`, `evaluate_generator` e `predict_generator`. Cada um desses métodos requer diferentes formas de geradores de dados.

  - `fit_generator` ⇒ esse método serve para treinar uma RNA com um gerador de dados, sendo que pode ser utilizado com dois geradores, uma para gerar os dados de treinamento e outro para gerar os dados de validação. Cada um desse geradores deve retornar uma `tupple` composta por (entradas, saídas) e consistem de instancias diferentes da mesma classe.

  - `evaluate_generator` ⇒ esse método serve para avaliar uma RNA após o treinamento e usa o mesmo gerador de dados usado no método `fit_generator`.

  - `predict_generator` ⇒ esse método serve para prever a saída de uma RNA já treinada e o gerador de dados usado nesse caso deve retornar somente dados de entrada, portanto é diferente dos usados nos métodos `fit_generator` e `evaluate_generator`.

- O gerador de dados mais simples do Keras é o da classe `ImageDataGenerator`, que serve principalmente para problemas de classificação de imagens.
  - Existem várias formas de usar esse gerador ⇒ uma forma é usá-lo é com o método `flow_from_directory`, que recebe um caminho para o diretório onde estão as imagens.
  - Esse gerador também permite realizar geração artificial de imagens (veremos isso na próxima aula).
- O gerador de imagens `ImageDataGenerator` do Keras está preparado para ser usado em três tipos de problema: (1) classificação binária, (2) classificação multiclasse, e (3) autoencoders.

Classificação binária

- Para usar o gerador `ImageDataGenerator` em um problema de classificação binária, as imagens devem estar em um diretório cuja estrutura é mostrada a seguir.

```
data/
     train/
           dogs/dog001.jpg
                dog002.jpg
                ...
           cats/cat001.jpg
                cat002.jpg
                ...
     validation/
                dogs/dog001.jpg
                     dog002.jpg
                     ...
                cats/cat001.jpg
                     cat002.jpg
                     ...
     test/
          dogs/dog001.jpg
               dog002.jpg
               ...
          cats/cat001.jpg
               cat002.jpg
               ...
```

  - Nesse exemplo, o diretório com os dados é estruturado para um problema de identificar se uma imagem mostra um gato ou um cachorro.
  - Nesse caso, o diretório onde estão os dados deve estar dividido em subdiretórios, cada um contendo as imagens de uma determinada classe.
  - O diretório principal, `data`, é subdividido em três diretórios, `train`, `validation` e `test`, e cada um desses diretórios é por sua vez subdividido em dois diretórios, `dogs` e `cats`, que contém respetivamente as imagens de cachorros e gatos.
  - Observe que os diretórios com as imagens de teste e de validação não são estritamente necessários para treinar uma RNA.

- Os nomes dos diretórios podem ser escolhidos como desejar e os nomes dos arquivos com as imagens não precisa ter o mesmo nome dos diretórios, nem a sequência numérica.
- O gerador carrega e cataloga as imagens de acordo com os nomes dos subdiretórios onde estão as imagens, bastando para isso que sejam nomes diferentes.

➢ O código a seguir mostra como criar um gerador de imagens usando a classe `ImageDataGenerator` do Keras. Esse gerador também normaliza as imagens dividindo os valores dos pixels por 255.

```
# Importa classe de geradores de imagens do Keras
from tensorflow.keras.preprocessing.image import ImageDataGenerator

# Cria um gerador que também normaliza as imagens
datagen = ImageDataGenerator(rescale=1./255)
```

- Antes de poder usar o gerador `ImageDataGenerator` ele deve ser importado da classe `tensorflow.keras.preprocessing.image`.
- Nesse exemplo, `datagen` é o nome escolhido para o gerador.
- O parâmetro `rescale` é responsável por fazer a normalização dos dados, ou seja, todos os exemplos são multiplicados por esse valor.

➢ Após criar o gerador devemos instanciá-lo e associá-lo a uma variável. Alguns parâmetros devem ser definidos quando instanciamos um gerador:

- Diretório onde estão as imagens ⇒ `'data/train'` (diretório onde estão as imagens de treinamento) e `'data/validation'` (diretório onde estão as imagens de validação);
- Dimensão das imagens ⇒ `target_size`;
- Tamanho do lote de treinamento ⇒ `batch_size`;
- Tipo de problema ⇒ `class_mode`.

O código a seguir mostra como instanciar os geradores de dados de treinamento e de validação, que carregam e pré-processam as imagens nos diretórios especificados.

```
# Instancia um gerador com as imagens de treinamento
train_generator = datagen.flow_from_directory(
    'data/train',
    target_size=(64, 64),
    batch_size=128,
    class_mode='binary')

# Instancia um gerador com as imagens de validação
validation_generator = datagen.flow_from_directory(
    'data/validation',
    target_size=(64, 64),
    batch_size=32
    class_mode='binary')
```

- O gerador de dados de treinamento é associado à variável `train_generator` e o de validação à variável `validation_generator`.
- Observe que `train_generator` e `validation_generator` são os nomes escolhidos para os dois geradores da mesma classe (`datagen`).
- Esses geradores apontam para os diretórios que contém os subdiretórios onde estão as imagens a serem utilizadas no treinamento.
- As imagens são redimensionadas para terem um tamanho de 64x64 pixels. Definir o tamanho da imagem no gerador tem a vantagem de poder testar a RNA com imagens de diferentes tamanhos com facilidade.
- O tamanho do lote (`batch_size`) não precisa ser o mesmo para as imagens de treinamento e validação.
- Existem outras formas de usar o `ImageDataGenerator` e muitos outros parâmetros que podem ser definidos ao criar um gerador de dados usando a classe `ImageDataGenerator` do Keras, para mais detalhes ver a documentação em https://www.tensorflow.org/api_docs/python/tf/keras/preprocessing/image/ImageDataGenerator.

➢ Finalmente tendo uma RNA para ser treinada para esse problema o seu treinamento é realizado com o método `fit_generator`, conforme mostrado no código a seguir. Lembre-se de que antes de ser treinada, a RNA precisa ser compilada com o método `compile`.

```
# Importa classe dos otimizadores
from tensorflow.keras import optimizers

# Compila a rede usando o método de otimização Adam
adam = optimizers.Adam(lr=0.001)
rna.compile(loss='binary_crossentropy', metrics=['accuracy'],
    optimizer=adam)

# Treina a RNA com o método fit_generator
history = rna.fit_generator(
    train_generator,
    steps_per_epoch=train_steps,
    epochs=50,
    validation_data=validation_generator,
    validation_steps=val_steps)
```

- `rna` é o nome dado para a RNA criada para resolver esse problema de classificação binária.
- `steps_per_epoch` = representa o número total de lotes de exemplos que devem ser gerados para finalizar uma época de treinamento. Tipicamente deve ser igual ao número total de exemplos de treinamento dividido pelo número de exemplos em cada lote (`batch_size`). Se tiver um conjunto de dados fixo esse parâmetro pode ser ignorado, mas é importante quando se usa geração artificial de dados ou quando se tem um número muito grande de dados e se quer usar somente uma parte dos dados. Por exemplo, se tivermos 1024 imagens de treinamento e o tamanho do lote (`batch_size`) for 128, então são necessários 8 passos (`steps_per_epoch`) para carregar todas as imagens.

- `validation_steps` = similar ao parâmetro `steps_per_epoch`, mas é referente aos dados de validação.
- Os resultados do processo de treinamento (função de custo e métrica) são guardados no dicionário `history`.

➢ Após treinar a RNA podemos avaliar o seu desempenho para os dados de teste usando o gerador de dados `datagen` com as imagens de teste e o método `evaluate_generator`. Para isso devemos primeiramente instanciar um novo gerador de dados com as imagens de teste. Essas duas operações são feitas conforme mostrado no código a seguir.

```
# Instancia um gerador com as imagens de teste
test_generator = datagen.flow_from_directory(
    'data/test',
    target_size=(64, 64),
    batch_size=32,
    class_mode='binary')

# Calcula a função de custo e a métrica para os dados de teste
custo_metrica_test = rna.evaluate_generator(test_generator,
    steps=test_steps)
```

- O gerador de imagens de teste para ser usado com o método `evaluate_generator` é o mesmo usado para o método `fit_generator` com as imagens de treinamento e validação.
- O parâmetro `steps` tem a mesma função do parâmetro `validation_steps` do comando `fit`.

➢ Finalmente podemos usar a RNA para fazer previsões usando o método `predict_generator`. Porém, o método `predict_generator` recebe como entrada somente as imagens e não as imagens e as suas classes, como nos casos dos métodos `fit_generator` e `evaluate_generator`. Dessa forma temos que criar um gerador de dados que retorna somente as imagens normalizadas com a dimensão correta.

➢ Um gerador de imagens para ser usado com o método `predict_generator` é criado da mesma forma que o usado nos métodos `fit_generator` e `evaluate_generator`, com a única diferença de que não se deve definir o tipo de problema, ou seja, o parâmetro `class_mode` deve ser igual a `None`. No código a seguir é mostrado como fazer isso.

```
# Instancia um gerador de previsão com as imagens de teste
generator = datagen.flow_from_directory(
        'data/test',
        target_size=(64, 64),
        batch_size=16,
        class_mode=None, # não se deve incluir a classe das imagens
        shuffle=False) # mantém dados na mesma ordem
```

➢ Após criar o gerador para previsão pode-se calcular as previsões da RNA para as imagens no diretório definido no gerador.

```
# Calcula a probabilidade da classe e depois determina a classe
prob = model.predict_generator(generator, steps=test_steps)
y_prev = np.around(prob)
```

- **Observação:**
  - Observe que o método `predict_generator` é feito para colocar uma RNA em operação e não para avaliar o seu desempenho.
  - Usar o método `predict_generator` para testar uma RNA em geral é mais complicado do que usar simplesmente o método `predict`, pois além de exigir criar um gerador de dados de teste, dificulta verificar o desempenho da RNA pelo fato de não se ter facilmente disponível as saídas reais desses dados.
  - O método `predict_generator` somente deve ser usado para avaliar uma RNA se existir problema de memória para carregas os dados de teste.

Classificação multiclasse

- O uso do gerador `ImageDataGenerator` em um problema de classificação multiclasse é muito similar ao que é usado em um problema de classificação binária.

- Por exemplo, para um problema de classificação multiclasse com 3 classes, tal como, verificar se uma imagem mostra um gato, ou um cão, ou um cavalo, as imagens devem estar em um diretório cuja estrutura é mostrada a seguir.

```
data/
     train/
           dogs/dog001.jpg
                dog002.jpg
                ...
           cats/cat001.jpg
                cat002.jpg
                ...
           horses/horse001.jpg
                  horse002.jpg
                   ...
     validation/
                dogs/dog001.jpg
                     dog002.jpg
                     ...
                cats/cat001.jpg
                     cat002.jpg
                     ...
                horses/horse001.jpg
                       horse002.jpg
                        ...
     test/
          dogs/dog001.jpg
               dog002.jpg
               ...
          cats/cat001.jpg
               cat002.jpg
               ...
          horses/horse001.jpg
                 horse002.jpg
                  ...
```

- O diretório principal, `data`, é subdividido em três diretórios, `train`, `validation` e `test`, e cada um desses diretórios é por sua vez subdividido em três diretórios, `dogs`, `cats` e `horses`, que contém respetivamente as imagens de cães, gatos e cavalos.
- Os nomes dos diretórios podem ser escolhidos como desejar e os nomes dos arquivos com as imagens não precisa ter o mesmo nome dos diretórios nem uma sequência numérica.
- O gerador carrega e cataloga as imagens de acordo com os nomes dos subdiretórios onde estão as imagens.
- Observe novamente que os diretórios com as imagens de teste e de validação não são estritamente necessários para treinar uma RNA.

➢ A instanciação do gerador de imagens usando a classe `ImageDataGenerator` do Keras para um problema de classificação multiclasse é feita praticamente da mesma forma como é feita para o problema de classificação binária.

➢ Após criar o gerador devemos instanciá-lo e associá-lo a uma variável, definindo o diretório onde estão as imagens, da mesma forma como foi feito para o problema de classificação binária.

➢ No código a seguir é mostrado como instanciar um gerador e associá-lo a uma variável para um problema de classificação multiclasse.

```
# Importa classe de geradores de imagens do Keras
from tensorflow.keras.preprocessing.image import ImageDataGenerator

# Cria um gerador que também normaliza as imagens
datagen = ImageDataGenerator(rescale=1./255)

# Instancia um gerador com as imagens de treinamento
train_generator = datagen.flow_from_directory(
    'data/train',
    target_size=(64, 64),
    batch_size=128,
    class_mode='categorical')

# Instancia um gerador com as imagens de validação
validation_generator = datagen.flow_from_directory(
    'data/validation',
    target_size=(64, 64),
    batch_size=32
    class_mode='categorical')
```

- Observe que a única diferença entre os geradores de dados usados para classificação binária e classificação multiclasse é a variável `class_mode`., que no caso de classificação multiclasse deve ser definida como sendo `'categorical'`.

- Finalmente, a compilação e treinamento de uma RNA para um problema de classificação multiclasse usando geradores de dados com o Keras é realizado com os métodos `compile` e `fit_generator` exatamente igual a como foi feito para o problema de classificação binária.

6. Construção de um gerador de dados para usar com o Keras

- Apesar do TensorFlow-Keras fornecer geradores de dados eficientes, esses geradores tem recursos limitados. A razão disso, é que cada novo problema exige dados diferentes e, portanto, exige um gerador que realiza diferentes processamentos e, principalmente, que gera dados de saídas não convencionais.

- Por exemplo, um problema pode exigir uma imagem com uma máscara especial, ou pode-se ter um problema onde a saída desejada é também uma imagem. Um problema desse tipo foi realizado no Trabalho #3 – Redes Complexas, onde a imagem de entrada é uma face com uma máscara cobrindo o nariz e a saída é a imagem original da face. Os exemplos desse conjunto de dados foram criados usando um gerador de dados.

- Nesses casos temos que criar o nosso próprio gerador. Criar um gerador específico para ser usado com o Keras não é difícil porque a estrutura de todos os geradores é sempre a mesma.

- O primeiro passo para criar um gerador para ser usado com o Keras é criar um gerador que inicialmente somente carrega as imagens e depois as normaliza e redimensiona. Esse gerador inicial é mostrado no código a seguir e tem as seguintes características:

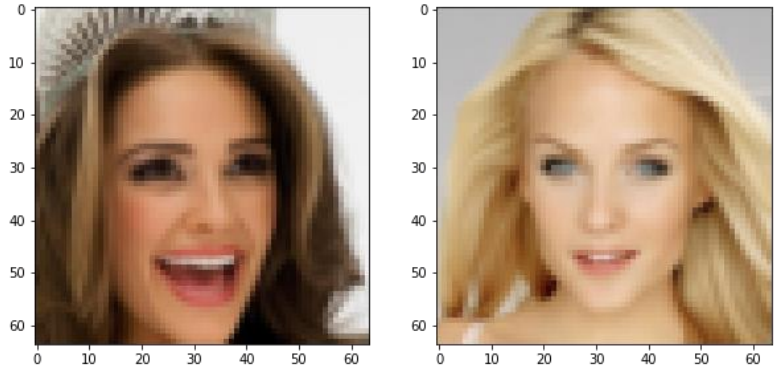
- Recebe como argumentos a lista de arquivos com as imagens desejadas (`img_paths`) e a dimensão desejada para as imagens (`img_size`).
- Os pixels das imagens são normalizados para valores entre 0 e 1, que é a normalização padrão usada para imagens.
- As imagens são redimensionadas para a dimensão `img_size`.

```
# Importa funções da biblioteca skimage
from skimage.io import imread
from skimage.transform import resize

# Cria gerador que carrega imagem, normaliza e redimensiona
def image_gen(img_paths, img_size=(512, 512, 3)):
    # Iterage em todos os caminhos das imagens
    for img_path in img_paths:

        # Carrega as imagens e normaliza entre 0 e 1
        img = imread(img_path) / 255.

        # Redimensiona as imagns
        img = resize(img, img_size, preserve_range=True)

        # Yield imagens
        yield img
```

- A biblioteca `skimage` possui várias funções para processamento de imagens, sendo uma delas a função `resize` para mudar a dimensão de uma imagem. Para mais detalhes sobre essa biblioteca ver https://scikit-image.org/.

➢ No código a seguir é mostrado como usar o gerador `image_gen` para carregar, normalizar e redimensionar duas imagens.

```
# Importa biblioteca matplotlib
import matplotlib.pyplot as plt

# Define tamanho da imagem
img_size = (64, 64, 3)

# Instancia o gerador
ig = image_gen(train_img_paths, img_size)

# Testa o gerador 2 vezes
first_img = next(ig)
sec_img = next(ig)

# Mostra as imagens gerada
f, pos = plt.subplots(1, 2, figsize=(10, 10))
pos[0].imshow(sec_img)
pos[1].imshow(sec_masked_img)
```



- A variável `train_img_paths` contém os caminhos (diretório e nome dos arquivos) de todas as imagens de treinamento. Essa variável é definida conforme explicado na seção 4 (Gerador de imagens simples) usando funções das bibliotecas `os` e `glob`.

➢ O próximo passo é criar o gerador para ser usado pelo método `fit_generator` do Keras. Esse gerador de imagens é mostrado no código a seguir e tem as seguintes características:

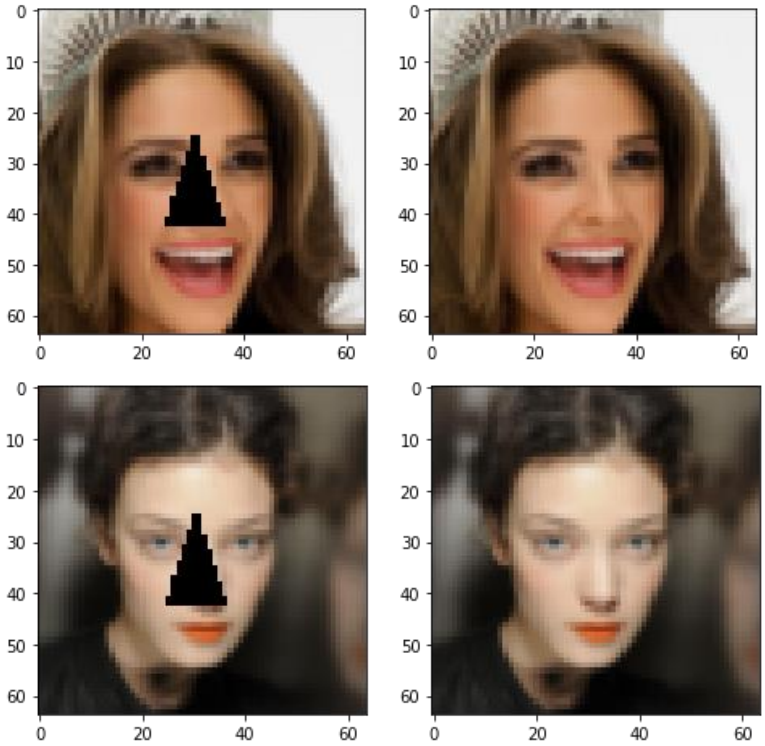
- Recebe como argumentos: (1) a lista de arquivos com as imagens desejadas (`img_paths`); (2) a imagem com a máscara (`mask`) usada para cobrir o nariz; (3) a dimensão desejada para as imagens (`img_size`); e (4) o tamanho desejado para os lotes (`batchsize`).
- A máscara é sobre imposta na imagem e as imagens com a máscara escondendo o nariz (`masked_img`) e a original (`img`) são retornadas.
- Diferença básica de um gerador para ser usado com o Keras e um gerador comum, é a presença de um loop infinito que somente termina no final de uma época.
- As instruções de atribuição de variáveis no Python não copiam objetos, elas criam ligações entre o objeto destino e o original, assim, quando alteramos um deles, o outro se altera automaticamente. Às vezes, é necessária uma cópia para que se possa alterá-la sem alterar o original, nesse caso se usa a biblioteca `copy`. Para mais detalhes sobre essa biblioteca ver https://docs.python.org/3/library/copy.html.
- O método `append` aplicado para um tensor numpy adiciona um novo tensor no final de uma lista. A operação `append` cria um novo tensor e descarta o anterior. Todas as dimensões dos tensores da lista devem ser as mesmas.
- A função `stack` da biblioteca numpy concatena uma sequência de tensores ao longo de um eixo definido pelo parâmetro `axis`.

```
# Importa funções da biblioteca skimage e função copy
from skimage.transform import resize
import copy

# Cria gerador para ser usado com o Keras
def face_batch_generator(img_paths, mask, img_size, batchsize):

    # Inicializa loop infinito que termina no final do treinamento
    while True:
        # Instancia gerador de imagens image_gen
        ig = image_gen(img_paths, img_size)

        # Inicializa lista de imagens com máscara e sem máscara
        batch_masked_img, batch_img = [], []

        # Redimensiona imagem da máscara
        mask = resize(mask, img_size, preserve_range=True)
        # Carrega imagens da lista e adiciona máscara até atingir o
        número de imagens igual a batchsize
        for img in ig:
            # Sobrepõe máscara na imagem
            masked_img = copy.deepcopy(img)
            masked_img[mask==0] = 0

            # Adiciona imagem com máscara e imagem original na lista
            batch_masked_img.append(masked_img)
            batch_img.append(img)

            # Se atingir o tamanho do lote retorna lote
            if len(batch_img) == batchsize:
                yield np.stack(batch_masked_img, axis=0),
                    np.stack(batch_img, axis=0)
                batch_masked_img, batch_img = [], []

            # Se ainda tiver lote meio vazio, retorna lote
            if len(batch_img) != 0:
            yield np.stack(batch_masked_img, axis=0),
                np.stack(batch_img, axis=0)
            batch_masked_img, batch_img = [], []
```

- No código a seguir é mostrado como usar o gerador `face_image_gen` para gerar dois exemplos de treinamento compostos por imagens de faces com a máscara tapando o nariz (dado de entrada) e as imagens originais das mesmas faces (dado de saída).
  - A primeira operação é carregar a imagem da máscara que está no arquivo `mask.jpg` e transformá-la em um tensor numpy.
  - Após definir a dimensão desejada para as imagens e instanciar o gerador ele pode ser usado para criar os exemplos de treinamento com o comando `next`.

```
# Carrega arquivo com a mascara e transforma em tensor numpy
mask = imread("mask.jpg")
mask = np.array(mask)

# Define dimensão das imagens
img_size = (64,64,3)

# Instancia o gerador
train_datagen = face_batch_generator(train_img_paths, mask,
    img_size, batchsize=32)

# Usa o gerador duas vezes
first_img_masked_batch, first_img_batch = next(train_datagen)
sec_img_masked_batch, sec_img_batch = next(train_datagen)

# Apresenta dimensão dos tensores de entrada de saída
print('Dimensão dos tensores de imagens =',
    first_img_masked_batch.shape, first_img_batch.shape)

# Mostra as imagens do primeiro exemplo dos lotes gerados
f, pos = plt.subplots(1, 2, figsize=(8, 8))
pos[0].imshow(first_img_masked_batch[0])
pos[1].imshow(first_img_batch[0])
plt.show()

f, pos = plt.subplots(1, 2, figsize=(8, 8))
pos[0].imshow(sec_img_masked_batch[0])
pos[1].imshow(sec_img_batch[0])
plt.show()
```

```
Dimensão dos tensores de imagens = (32, 64, 64, 3) (32, 64, 64, 3)
```



- Para usar esse gerador no treinamento de uma RNA basta instanciá-lo para os dados de treinamento e validação, conforme mostrado no código a seguir.

```
# Define tamanho do lote
BATCHSIZE = 32

# Instancia os geradores de dados de treinamento e validação
traingen = image_batch_generator2(train_img_paths, mask, img_size,
    batchsize=BATCHSIZE)
valgen = image_batch_generator2(val_img_paths, mask, img_size,
    batchsize=BATCHSIZE)

# Define função para calcular número de lotes por época
def calc_steps(data_len, batchsize):
    return (data_len + batchsize - 1)//batchsize

# Calcula números de lotes por época
train_steps = calc_steps(len(train_img_paths), BATCHSIZE)
val_steps = calc_steps(len(val_img_paths), BATCHSIZE)
```

- A função `calc_steps` calcula o número de vezes que o gerador tem que ser chamado para carregar todos os exemplos dos conjuntos de dados de treinamento e validação.

- Tendo os geradores de dados de treinamento e validação instanciados, o treinamento da RNA é realizado com o método `fit_generator` da mesma forma como é feito para o caso de se usar o gerador `ImageDataGenerator` do Keras, conforme mostrado no código a seguir.

```
# Treinamento da rede
history = model.fit_generator(
    traingen, # nome do gerador de dados de treinamento
    steps_per_epoch=train_steps,
    epochs=20,
    validation_data=valgen, # nome do gerador de dados de validação
    validation_steps=val_steps,
    verbose=1)
```

7. Camada Lambda

- Em algumas situações é necessário realizar cálculos simples dentro de uma RNA. Nesses casos podemos usar uma camada tipo `Lambda`.

- Funções tipo `Lambda` consistem de uma estrutura natural da linguagem Python, que consiste de funções simples e restritas, que não precisam ter nome, definidas em geral por uma única linha de programação.

- A ideia da função tipo `Lambda` do Python é recriada no Keras como sendo uma camada de uma RNA que realiza cálculos simples.

- Uma camada tipo `Lambda` pode ser inserida em qualquer posição de uma RNA.

- No código a seguir é apresentada uma camada tipo `Lambda` que calcula o quadrado dos elementos do tensor de entrada da camada para um modelo de RNA `Sequencial` e para um modelo de RNA `Functional` do Keras.

```
# Importa camada tipo Lambad do Keras
from tensorflow.keras import layers

# Adiciona camada Lambda em uma RNA sequencial
rna.add(layers.Lambda(lambda x: x**2))

# Adiciona camada Lambda em uma RNA funcional
square_layer = layers.Lambda(lambda x: x**2)
x2 = square_layer(x1)
```

  - No caso da camada `Lambda` do modelo `Functional`, `x1` é o tensor de entrada da camada `Lambda`.

- Outro exemplo de uma camada `Lambda` é a normalização dos dados de entrada que pode ser realizada por uma camada da RNA no lugar de serem feitos em uma etapa anterior de pré-processamento, ou por um gerador de dados.

- Uma camada `Lambda` para normalizar os dados de entrada pode ser vantajosa no caso de processamento de imagens, para economizar memória.

  - Nas imagens coloridas no padrão RGB cada pixel consiste de três números inteiros de 8 bits.
  - Para uma RNA processar uma imagem os valores dos pixels têm que ser normalizados, ou seja, devem ser transformados para valores reais entre 0 e 1.
  - Um número real ocupa no mínimo 32 bits, ou seja, ocupa pelo menos 4 vezes mais memória do que um pixel da imagem original, exigindo, dessa forma, mais memória para armazenar os dados.

- No código a seguir é mostrado um exemplo de como configurar uma RNA para uma tarefa de classificação binária com uma camada `Lambda` para normalizar os dados de entrada. Observe que a camada `Lambda` deve vir logo a pós a camada que define a entrada da RNA (camada tipo `Input`).

```
# Instancia a rede
rna = Sequential()

# Adiciona camada de entrada para receber os dados de entrada
rna.add(layers.Input(shape=data_shape))

# Adiciona camada Lambda para normalização
rna.add(layers.Lambda(lambda x: x/255.))

# Adiciona demais camadas convolucionais e densas
rna.add(layers.Conv2D(32, (3, 3), strides=1, activation='relu'))
rna.add(layers.MaxPooling2D((2, 2)))
rna.add(layers.Conv2D(64, (3, 3), strides=1, activation='relu'))
rna.add(layers.MaxPooling2D((2, 2)))
rna.add(layers.Flatten())
rna.add(layers.Dense(256, activation='relu'))
rna.add(layers.Dense(1, activation='sigmoid'))
```

- Essa RNA é do tipo sequencial e possui duas camadas convolucionais, duas camadas de maxpooling e duas camadas densas.
- Como se tem um problema de classificação binária, a camada de saída possui somente um neurônio com função de ativação sigmoid.
- Antes da primeira camada densa tem-se a camada `Flatten` para ajustar a dimensão da saída da camada maxpooling à entrada da camada densa.