# CNN

A classificação de imagens é um problema importante em áreas como reconhecimento de padrões, visão computacional e inteligência artificial.

As Redes Neurais Convolucionais (CNNs) são uma das arquiteturas mais populares para resolver problemas de classificação de imagens hoje em dia. Elas são capazes de extrair características significativas das imagens e usá-las para classificar novas imagens.

Neste artigo, vamos explicar de forma didática como é montada uma CNN para resolver um problema de classificação binária. Vamos abordar conceitos importantes como camadas de convolução, pooling, normalização, dropout e camadas densas. Além disso, vamos discutir como preparar os dados para treinar uma CNN e como avaliar sua performance.

Ao final do artigo, vamos apresentar uma breve explicação para quem estiver interessado em treinar uma CNN multiclasse.

Esperamos que este artigo seja útil para quem está iniciando seus estudos em CNNs e que possa ajudar a compreender melhor como elas funcionam e como podem ser aplicadas em problemas reais.

## 1° PASSO: DADOS

### 1° Subpasso: Coletar as imagens

Comece coletando imagens que representem bem as classes que você deseja identificar com a CNN. Neste caso, são duas classes: vacas e falta de vacas. Você pode coletar essas imagens de diferentes fontes, como uma câmera, banco de dados público ou da internet em geral.

### 2° Subpasso: Tratar as imagens

Para treinar uma CNN, é necessário preparar as imagens antes. Primeiramente, é preciso redimensioná-las para o tamanho adequado (144x177 pixels), normalizar as intensidades de pixel e, se necessário, aplicar técnicas de processamento de imagem, como desfoque ou aumento de contraste. Para facilitar esse processo, há um código chamado " normalização_das_imagens.py" disponível no repositório que fará esse trabalho automaticamente.

Outra etapa importante é rotular as imagens, ou seja, atribuir uma classe ou categoria a cada uma delas. No caso de identificar vacas, as imagens com vacas devem ser rotuladas com o número 1 e as sem vacas com o número 0. Isso permitirá que o modelo saiba qual é a classe correta para cada imagem durante o treinamento.

Embora seja possível fazer o processo de rotulagem manualmente, é mais eficiente atribuir as classes ao mesmo tempo que as imagens são preparadas para o treinamento.

**3° Subpasso: Transformar as imagens em Numpy “.npz”**

O terceiro passo é transformar as imagens tratadas em matrizes Numpy e salvá-las em arquivos ".npz". Para fazer isso, você pode usar a biblioteca NumPy em Python, que fornece funções para carregar e salvar matrizes Numpy em diferentes formatos. “conversor_numpy.py”, arquivo também se encontra no repositório git.

Para transformar as imagens tratadas em matrizes Numpy e salvá-las em arquivos ".npz", podemos utilizar a biblioteca NumPy em Python. No código fornecido, há um exemplo de como fazer isso. “conversor_numpy.py”.

Primeiramente, definimos o diretório onde estão as imagens tratadas e o diretório onde serão salvos os arquivos ".npz". Em seguida, definimos as classes que queremos identificar, que no exemplo é apenas "vaca", portanto, além de definir a classe "vaca", também é necessário definir a classe "sem vaca" e atribuir o rótulo 0 a ela. Dessa forma, a CNN poderá distinguir entre as duas classes durante o treinamento.

Depois, percorremos cada imagem de cada classe e transformamos a imagem em uma matriz Numpy. Armazenamos as matrizes em uma lista de dados e os rótulos em outra lista.

Por fim, convertemos as listas em arrays Numpy e salvamos em um arquivo .npz. É importante lembrar que, caso o diretório de dados não exista, ele será criado automaticamente.

No caso do código de treinamento que estou fornecendo não e necessário fazer a separação entre dados de teste, validação e treinamento pois isso é feito automaticamente dentro do nosso treinamento.

## 2° PASSO

**Treinar a Rede Convolucional:** Para isso vamos entender oque o nosso código está fazendo.

## IMPORTAR BIBLIOTECAS NECESSARIAS

```
# Importar bibliotecas necessárias
import os
import numpy as np
import tensorflow as tf
from sklearn.model_selection import train_test_split
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Conv2D, MaxPooling2D, Flatten, D
ense, Dropout, Activation
from tensorflow.keras.preprocessing.image import ImageDataGenerator
import matplotlib.pyplot as plt
```

Nessa parte, as bibliotecas necessárias são importadas. As bibliotecas incluem o TensorFlow, que é uma plataforma de aprendizado profundo, e outras bibliotecas usadas para visualização e manipulação de dados.

## VERIFICAR DISPONIBILIDADE DA GPU

```
# Verificar se a GPU está disponível
physical_devices = tf.config.list_physical_devices('GPU')
if len(physical_devices) > 0:
    tf.config.experimental.set_memory_growth(physical_devices[0], Tr
ue)
    print('GPU disponível:', physical_devices[0])
else:
    print('Nenhuma GPU disponível.')

tf.test.gpu_device_name()
```

Essas linhas de código verificam se uma GPU está disponível e definem algumas configurações para permitir que o TensorFlow use toda a memória da GPU disponível.

O uso de GPU (unidade de processamento gráfico) no treinamento de redes neurais pode trazer várias vantagens, incluindo velocidade, eficiência energética, escalabilidade e flexibilidade. As GPUs são projetadas para processar grandes quantidades de dados em paralelo, o que acelera o treinamento da rede e reduz o consumo de energia. Além disso, várias GPUs podem ser usadas juntas para acelerar ainda mais o processo de treinamento.

## DEFINIR PARÂMETROS

```
# Definir parâmetros
batch_size = 32
epochs = 20
learning_rate = 0.001
```

Essas variáveis definem os parâmetros para o treinamento do modelo, incluindo o tamanho do lote, o número de épocas e a taxa de aprendizado.

Batch size, epochs e learning rate são parâmetros importantes no treinamento de redes neurais, incluindo redes neurais convolucionais (CNNs).

**Batch size** refere-se ao número de amostras de treinamento que são utilizadas em cada iteração do algoritmo de treinamento. Em outras palavras, é o número de exemplos de treinamento que são processados em cada etapa de backpropagation. O uso de um batch size maior pode acelerar o processo de treinamento, mas também pode exigir mais memória e recursos de computação.

**Epochs** refere-se ao número de vezes que todo o conjunto de dados de treinamento é passado pela rede durante o processo de treinamento. Cada época é composta por várias iterações, onde cada iteração processa um batch de amostras. O número de epochs deve ser escolhido cuidadosamente para evitar overfitting ou underfitting da rede.

**Learning rate** refere-se à taxa na qual a rede ajusta seus pesos durante o processo de treinamento. É um hiperparâmetro crítico que controla a magnitude das atualizações de peso em cada etapa do treinamento. Um learning rate muito alto pode levar a oscilações e instabilidades no processo de treinamento, enquanto um learning rate muito baixo pode levar a um treinamento lento ou estagnado.

Em resumo, o batch size determina o número de amostras de treinamento processadas em cada iteração, epochs determina o número de vezes que todo o conjunto de dados de treinamento é passado pela rede e learning rate determina a taxa na qual a rede ajusta seus pesos durante o processo de treinamento. Esses parâmetros devem ser ajustados cuidadosamente para garantir que a rede seja treinada de maneira eficiente e precisa.

## CARREGAR DADOS

```
# Carregar dados
diretorio_dados = '/home/carlin/Imagens/dados'
try:
    dados_vacas = np.load(os.path.join(diretorio_dados, 'vaca.np
z'))['dados']
    dados_sem_vacas = np.load(os.path.join(diretorio_dados, 'semva
ca.npz'))['dados']
except:
    print("Erro ao carregar os dados.")
```

Essas linhas de código carregam os dados de treinamento e teste que foram salvos em arquivos .npz. Existem dois arquivos .npz, um contém imagens de vacas e outro contém imagens sem vacas.

## RÓTULOS E CONCATENAR DADOS

```
# Adicionar labels aos dados
x_vacas = dados_vacas
y_vacas = np.ones(len(x_vacas), dtype=np.int32) # 1 representa vacas
x_sem_vacas = dados_sem_vacas
y_sem_vacas = np.zeros(len(x_sem_vacas), dtype=np.int32) # 0 represe
nta falta de vacas

# Concatenar dados
x = np.concatenate((x_vacas, x_sem_vacas), axis=0)
y = np.concatenate((y_vacas, y_sem_vacas), axis=0)
```

Estas linhas do código são responsáveis por preparar os dados para o treinamento de uma rede neural.

Na primeira parte, o código adiciona rótulos **(labels)** aos dados de treinamento. Aqui, o conjunto de dados é dividido em duas partes**: "dados_vacas"** e **"dados_sem_vacas"**. A variável "x_vacas" recebe os dados referentes às vacas, enquanto a variável "y_vacas" recebe um array de 1s do mesmo tamanho de "x_vacas". Isso significa que cada amostra de "x_vacas" é rotulada como uma vaca. De maneira similar, a variável "x_sem_vacas" recebe os dados referentes à falta de vacas, enquanto a variável "y_sem_vacas" recebe um array de 0s (zeros) do mesmo tamanho de "x_sem_vacas", indicando que cada amostra em "x_sem_vacas" não contém uma vaca.

Na segunda parte, os dados são concatenados em um único conjunto. A variável "x" é criada pela concatenação das variáveis "x_vacas" e "x_sem_vacas" ao longo do eixo 0 (verticalmente). Da mesma forma, a variável "y" é criada pela concatenação das variáveis "y_vacas" e "y_sem_vacas" ao longo do eixo 0.

Esses passos são importantes para garantir que os dados estejam corretamente rotulados e preparados para serem usados no treinamento da rede neural.

## DIVIDIR DADOS EM CONJUNTO DE TREINO E TESTE

```
# Dividir 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)
```

Essas linhas de código dividem os dados em conjuntos de treinamento e teste, com uma proporção de 80% para treinamento e 20% para teste.

### POR QUE É MELHOR FAZER ESSA DIVISÃO DENTRO DO PROCESSO?

Dividir os dados de treinamento e teste dentro do processo de treinamento, em vez de usar a abordagem convencional de dividir os dados antes do treinamento, pode trazer várias vantagens:

Utilização mais eficiente dos dados: dividir os dados dentro do processo de treinamento permite que a rede neural utilize todos os dados disponíveis para treinamento e teste. Isso é especialmente importante para conjuntos de dados pequenos, onde uma divisão convencional pode resultar em um conjunto de treinamento muito pequeno para uma boa generalização.

Evita o vazamento de informações: na abordagem convencional de divisão de dados, é possível que informações do conjunto de teste vazem para o conjunto de treinamento durante o processo de desenvolvimento do modelo. Isso pode levar a uma superestimação da precisão do modelo. Dividir os dados dentro do processo de treinamento evita esse problema, pois o conjunto de teste é mantido completamente separado durante todo o processo de treinamento.

Melhor ajuste de hiperparâmetros: ao dividir os dados dentro do processo de treinamento, é possível ajustar os hiperparâmetros do modelo (como a taxa de aprendizado e o número de épocas) de forma mais precisa e eficiente. Isso porque a rede neural é treinada e avaliada várias vezes com diferentes configurações de hiperparâmetros, permitindo que os ajustes sejam feitos com base nos resultados de validação.

Em resumo, dividir os dados de treinamento e teste dentro do processo de treinamento pode levar a uma utilização mais eficiente dos dados, evitar o vazamento de informações e permitir um melhor ajuste de hiperparâmetros. Essa abordagem pode resultar em modelos mais precisos e generalizados.

## NORMALIZAR DADOS

```
# Normalizar dados
x_train = x_train.astype('float32') / 255.0
x_test = x_test.astype('float32') / 255.0
```

Essas linhas de código normalizam os valores dos pixels das imagens de 0 a 255 para um intervalo de 0 a 1.

A normalização dos dados é uma etapa crítica no pré-processamento de dados para treinamento de redes neurais, incluindo redes neurais convolucionais (CNNs). A normalização refere-se ao processo de ajustar os valores das características (features) em uma escala comum, a fim de tornar o treinamento da rede mais eficiente e preciso.

## Importância da normalização de dados

Melhora a convergência: a normalização dos dados pode ajudar a garantir que a função de custo (loss function) converga mais rapidamente durante o treinamento, o que pode acelerar o processo de treinamento e levar a modelos mais precisos.

Reduz a sensibilidade a valores extremos: a normalização dos dados pode reduzir a sensibilidade da rede a valores extremos em determinadas características, que podem afetar negativamente o desempenho da rede.

Garante a estabilidade dos gradientes: a normalização dos dados pode ajudar a garantir que os gradientes calculados durante o treinamento sejam mais estáveis, o que pode ajudar a evitar problemas como o desaparecimento dos gradientes (vanishing gradients) e explosão dos gradientes (exploding gradients).

Ajuda a comparar diferentes características: a normalização dos dados pode ajudar a comparar diferentes características em uma escala comum, o que pode ser útil quando há características com diferentes escalas e unidades de medida.

No código fornecido, a normalização é realizada dividindo os dados por 255.0, que é o valor máximo que um pixel pode ter em uma imagem de 8 bits. Isso significa que os valores das características são ajustados para uma escala entre 0 e 1, o que pode ajudar a melhorar a eficiência e precisão do treinamento da rede.

## DIMENSÃO EXTRA PARA NÚMERO DE CANAIS

```
# Adicionar dimensão para número de canais
x_train = np.expand_dims(x_train, axis=-1)
x_test = np.expand_dims(x_test, axis=-1)
```

Essas linhas de código adicionam uma dimensão extra às imagens para indicar o número de canais (1 no caso de imagens em escala de cinza).

Adicionar uma dimensão extra para o número de canais é uma etapa importante no pré-processamento de dados para o treinamento de redes neurais convolucionais (CNNs), especialmente quando os dados de entrada são imagens RGB.

A maioria das CNNs é projetada para receber imagens com três canais (vermelho, verde e azul) como entrada. No entanto, os conjuntos de dados podem ter imagens com apenas um canal (escala de cinza) ou mais de três canais (por exemplo, imagens multiespectrais ou hiperespectrais).

Ao adicionar uma dimensão extra para o número de canais, o código está garantindo que todas as imagens de entrada tenham a mesma dimensão de canais, independentemente do número de canais original da imagem. No caso de imagens RGB, a dimensão extra adicionada é apenas uma repetição do mesmo canal para cada pixel da imagem.

O método "np.expand_dims" é usado para adicionar a dimensão extra ao final do array de dados, indicado pelo parâmetro "axis=-1". Isso cria uma nova dimensão no final do array, que será usada para armazenar o número de canais da imagem.

Em resumo, adicionar uma dimensão extra para o número de canais é uma etapa importante no pré-processamento de dados para o treinamento de CNNs, garantindo que todas as imagens de entrada tenham a mesma dimensão de canais e possam ser processadas eficientemente pela rede.

## DEFINIR MODELO

```
# Definir modelo
model = Sequential()
model.add(Conv2D(32, (3, 3), padding='same', input_shape=(144, 177,
1)))
model.add(Activation('relu'))
model.add(MaxPooling2D(pool_size=(2, 2)))
model.add(Dropout(0.25))

model.add(Conv2D(64, (3, 3), padding='same'))
model.add(Activation('relu'))
model.add(Conv2D(64, (3, 3)))
model.add(Activation('relu'))
model.add(MaxPooling2D(pool_size=(2, 2)))
model.add(Dropout(0.25))

model.add(Flatten())
model.add(Dense(512))
model.add(Activation('relu'))
model.add(Dropout(0.5))
model.add(Dense(1))
model.add(Activation('sigmoid'))
```

Essas linhas de código definem a arquitetura do modelo de rede neural convolucional usando a API Keras do TensorFlow. O modelo inclui várias camadas convolucionais, de pooling, totalmente conectadas e de saída.

## Camada de convolução 2D (Conv2D):

A camada de convolução 2D realiza uma operação matemática que aplica um conjunto de filtros a uma imagem de entrada, a fim de extrair características importantes da imagem. Os filtros deslizam pela imagem e realizam uma multiplicação entre a região da imagem coberta pelo filtro e os valores dos pesos do filtro. O resultado é uma matriz de características que representa as características da imagem que foram detectadas pelo filtro. No modelo fornecido, a primeira camada Conv2D tem 32 filtros com tamanho de 3x3.

## Função de ativação ReLU (Activation 'relu'):

A função de ativação ReLU (Retified Linear Unit) é uma função matemática que introduz não-linearidade na rede. A função ReLU aplica a seguinte equação: f(x) = max(0, x). Isso significa que, se o valor de entrada for negativo, a saída da função será 0, caso contrário, a saída será o mesmo valor de entrada. A função ReLU é amplamente usada em redes neurais devido à sua simplicidade e eficácia na introdução de não-linearidade.

## Camada de pooling máxima 2D (MaxPooling2D):

A camada de pooling máxima 2D é usada para reduzir a dimensão espacial da saída da camada de convolução. A operação de pooling máxima seleciona o valor máximo em uma janela de pixels (por exemplo, 2x2) e usa esse valor como representante da região. Isso reduz a quantidade de parâmetros da rede e aumenta sua capacidade de generalização.

## Camada de dropout (Dropout):

A camada de dropout é usada para reduzir o overfitting, que é quando a rede se ajusta demais aos dados de treinamento e não consegue generalizar bem para novos dados. A camada de dropout desativa aleatoriamente algumas unidades da rede durante o treinamento, o que ajuda a rede a não se concentrar em características específicas do conjunto de treinamento e a aprender representações mais robustas e generalizadas. No modelo fornecido, a camada de dropout é definida com uma taxa de dropout de 0,25.

## Camada de achatamento (Flatten):

A camada de achatamento é usada para transformar a saída da camada anterior em uma única dimensão, achatando a matriz de características para que possa ser processada por uma camada densa. Essa camada é usada sempre que a saída de uma camada convolucional ou de pooling for passada para uma camada densa.

## Camada densa (Dense):

A camada densa executa uma operação de multiplicação de matriz entre a entrada e uma matriz de pesos, seguida de uma função de ativação. A camada densa é usada para aprender relações mais complexas entre as características extraídas anteriormente. No modelo fornecido, a camada densa tem 512 unidades e usa a função de ativação ReLU.

Para cada conjunto de camadas, os valores podem mudar para ajustar a rede à tarefa específica de classificação de imagem. O número de filtros, tamanho do kernel, taxa de dropout e número de unidades na camada densa podem ser ajustados para melhorar o desempenho da rede. O tamanho do kernel e o tamanho das janelas de pooling afetam a resolução espacial da saída da camada, enquanto o número de filtros afeta a quantidade de características extraídas. O número de unidades na camada densa afeta a complexidade da rede e sua capacidade de aprender relações mais complexas. Em resumo, os valores das camadas são ajustados para otimizar a rede para a tarefa de classificação de imagem específica.

## COMPILAR MODELO

```
# Compilar modelo
model.compile(loss='binary_crossentropy',
              optimizer=tf.keras.optimizers.Adam(lr=learning_rate),
              metrics=['accuracy'])
```

Essas linhas de código compilam o modelo definido anteriormente com uma função de perda binária, um otimizador Adam e a métrica de precisão.

Essas linhas de código compilam o modelo definido anteriormente, especificando a função de perda, o otimizador e as métricas usadas para avaliar o desempenho do modelo durante o treinamento.

**A função de perda (loss**) é uma medida da diferença entre as previsões da rede e as verdadeiras classes associadas às imagens de treinamento. No modelo fornecido, a função de perda é a 'binary_crossentropy', que é comumente usada em problemas de classificação binária.

**O otimizador (optimizer)** é o algoritmo usado para ajustar os pesos da rede durante o treinamento, com o objetivo de minimizar a função de perda. No modelo fornecido, o otimizador é o Adam, que é um algoritmo de otimização popular para redes neurais.

**As métricas (metrics)** são usadas para avaliar o desempenho da rede durante o treinamento. A métrica especificada no modelo fornecido é 'accuracy', que é a proporção de imagens classificadas corretamente em relação ao total de imagens.

Assim, ao compilar o modelo, estamos definindo a maneira como a rede será treinada e avaliada, especificando a função de perda, o otimizador e as métricas a serem utilizadas.

## GERADORES DE IMAGENS

```
# Criar geradores de imagem para aumentar o conjunto de dados de tre
inamento
    #Aplicando rotação, zoom, deslocamento horizontal e vertical e f
lip horizontal.
train_datagen = ImageDataGenerator(rotation_range=20,
                                   zoom_range=0.2,
                                   width_shift_range=0.2,
                                   height_shift_range=0.2,
                                   horizontal_flip=True)
train_generator = train_datagen.flow(x_train, y_train, batch_size=ba
tch_size)
```

Essas linhas de código criam geradores de imagem para aumentar o conjunto de dados de treinamento. As transformações aplicadas incluem rotação, zoom, deslocamento horizontal e vertical e flip horizontal.

## TREINAR MODELO

```
# Treinar modelo
history = model.fit(train_generator,
                    steps_per_epoch=len(x_train) // batch_size,
                    epochs=epochs,
                    validation_data=(x_test, y_test))
```

Esta parte do código treina o modelo na base de dados de treinamento. O método fit é usado para treinar o modelo. Ele recebe como entrada o gerador de dados de treinamento train_generator, que fornece imagens e rótulos de forma aleatória em cada época de treinamento. O parâmetro steps_per_epoch específica quantos lotes de treinamento devem ser executados em cada época. O número de épocas de treinamento é especificado pelo parâmetro epochs. O conjunto de dados de teste é

passado para o parâmetro validation_data, que é usado para avaliar o modelo durante o treinamento e para detectar possíveis problemas de overfitting. A função retorna um objeto history que contém as métricas do modelo ao longo do treinamento.

## AVALIAR MODELO

```
# Avaliar modelo
score = model.evaluate(x_test, y_test, verbose=0)
print('Test loss:', score[0])
print('Test accuracy:', score[1])
```

Esta parte do código avalia o modelo na base de dados de teste. O método evaluate é usado para avaliar o modelo. Ele recebe como entrada o conjunto de dados de teste x_test e seus rótulos correspondentes y_test. O parâmetro verbose é definido como 0 para evitar a impressão do progresso do teste na tela. A função retorna a perda e a acurácia obtidas no conjunto de teste, que são atribuídas à variável score. Essas métricas são então impressas na tela com a função print.

## PLOTAR ACURÁCIA E PERDA

```
# Plotar acurácia e perda ao longo das épocas
plt.plot(history.history['accuracy'], label='accuracy')
plt.plot(history.history['val_accuracy'], label='val_accuracy')
plt.plot(history.history['loss'], label='loss')
plt.plot(history.history['val_loss'], label='val_loss')
plt.xlabel('Epoch')
plt.ylabel('Accuracy/Loss')
plt.legend(loc='lower right')
plt.show()
```

Esta parte do código plota as curvas de acurácia e perda ao longo das épocas. A biblioteca matplotlib é usada para plotar as curvas. As curvas de acurácia de treinamento (accuracy), acurácia de teste (val_accuracy), perda de treinamento (loss) e perda de teste (val_loss) são plotadas em um gráfico. As curvas são rotuladas e os eixos do gráfico são etiquetados. A função show é usada para exibir o gráfico.

## CONFIRMAÇÃO DE TREINAMENTO E AVALIAÇÃO DO MODELO

```
# Mensagem de confirmação
print("O treinamento e avaliação do modelo foram concluídos com suce
sso!")
```

Por fim esta parte do código imprime uma mensagem indicando que o treinamento e a avaliação do modelo foram concluídos com sucesso. A mensagem é impressa na tela com a função print.

O seu modelo treinado estará no diretório especificado com nome definido “modeo.h5”

## 3 PASSO: TESTE

Para testar o seu modelo de rede neural convolucional, você pode usar o código pronto “testeCNN.py”. Este código carrega um modelo previamente treinado, processa imagens de teste e faz uma previsão da classe de cada imagem.

## CÓDIGO “testeCNN.py”:

Este código primeiro carrega o modelo previamente treinado usando o método load_model da biblioteca keras. Em seguida, ele carrega seis imagens de teste usando a biblioteca PIL e as redimensiona para o tamanho esperado pela rede neural. As imagens são então convertidas em arrays numpy usando o método img_to_array da biblioteca keras.preprocessing.image. Os arrays são expandidos para ter forma (1, 144, 177, 1) usando o método expand_dims da biblioteca numpy e normalizados dividindo por 255.0.

O código então faz uma previsão das classes das imagens usando o método predict do modelo treinado, que retorna as probabilidades de cada imagem pertencer à classe positiva. Finalmente, o código imprime uma mensagem indicando se cada imagem é ou não uma vaca, com base na probabilidade de cada imagem ser da classe positiva.

## ATERAÇOES PARA FUNÇÃO DE MULTICLASSE:

Quando estamos trabalhando com classificação de imagens, é comum que haja mais de duas classes. No código fornecido, o modelo está preparado para lidar com somente duas classes, que são "vaca" e "sem vaca".

Se quisermos adicionar mais classes, precisamos fazer algumas alterações no código. Em primeiro lugar, precisamos alterar a camada de saída da rede neural para ter um número de neurônios igual ao número de classes desejado. Por exemplo, se houver 3 classes, a camada de saída deve ter 3 neurônios.

Além disso, precisamos alterar a função de ativação na camada de saída para "softmax", que é adequada para problemas multiclasse. A função de perda também precisa ser alterada para "categorical_crossentropy", que é adequada para problemas multiclasse.

É importante lembrar que os dados também precisam ser alterados para incluir as classes adicionais. Isso significa que precisamos alterar a forma como os dados são carregados e rotulados para incluir as classes adicionais. Além disso, precisamos garantir que haja exemplos suficientes para cada classe em ambos os conjuntos de treinamento e teste.

Para garantir que todos os pixels estejam na mesma escala, precisamos alterar a forma como os dados são normalizados. Também precisamos alterar a forma como os dados são gerados com o ImageDataGenerator para incluir a opção "class_mode='categorical'" para a criação de rótulos no formato one-hot encoding.

Ao avaliar o modelo, precisamos incluir a métrica "categorical_accuracy" para avaliar a precisão do modelo em problemas multiclasse. E, por fim, precisamos alterar a forma como a acurácia e a perda são plotadas para refletir a nova métrica de avaliação.

Espero que isso ajude a entender as mudanças necessárias ao lidar com problemas de classificação multiclasse em redes neurais.

Carlos Silva

Git: Radagon/Hefesto66

Estudante da aréa de Machine learning