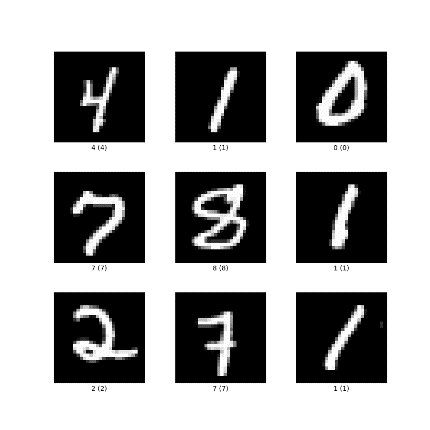
Roteiro de Redes Neurais e Deep Learning

Este roteiro trata dos principais comandos relacionados à *Redes Neurais* e *Redes Convolucionais* utilizando Python e uma das bibliotecas, que é Tensor Flow Keras. Para esta prática, vamos utilizar 2 (dois) dataset: Mnist e Cifar10. A seguir, uma breve descrição destes.

**MNIST**

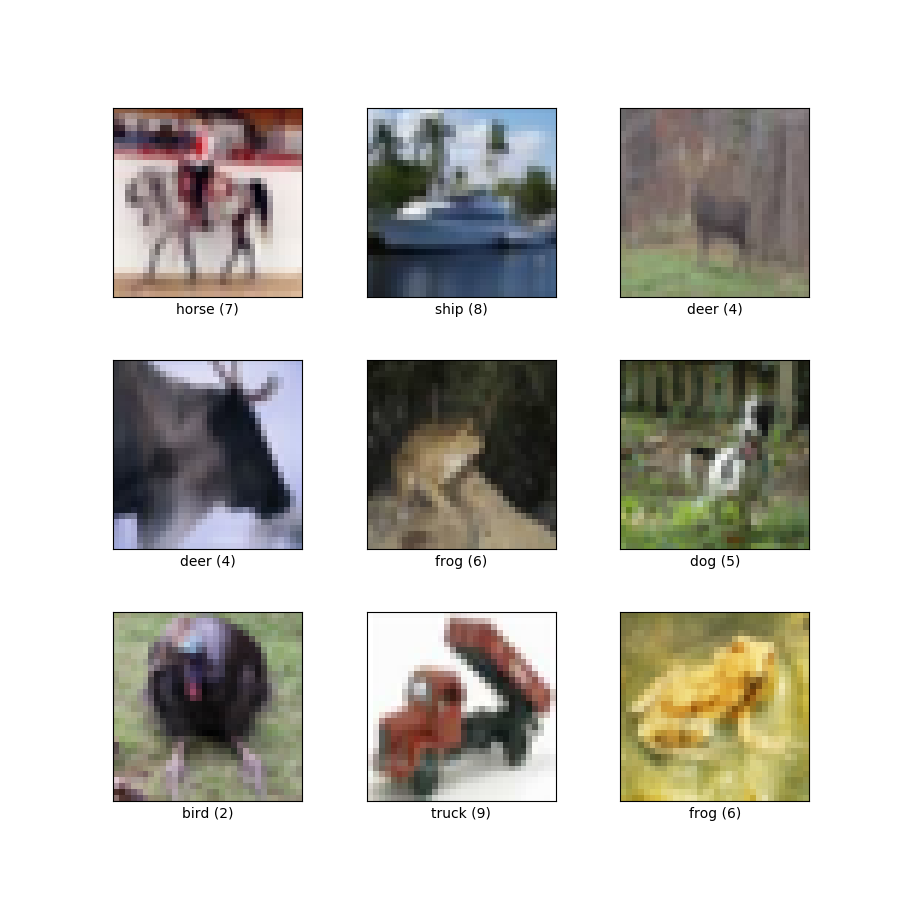
Dataset composto por imagens 28x28 pixels de dígitos 0 a 9 escritos a mão. O conjunto total tem 60 mil imagens para treinamento e 10 mil imagens para teste. Um exemplo das imagens a seguir.



As imagens são rotuladas de 0 a 9, total de 10 classes.

**CIFRA10**

Dataset composto por imagens 32x32 pixels de objetos diversos, em 10 classes mutuamente exclusivas sem sobreposição, sendo elas: airplane, automobile, bird, cat, deer, dog, frog, horse, ship e truck. O conjunto total tem 60 mil imagens, sendo 6 mil por classe. O conjunto é dividido em 50 mil imagens para treinamento e 10 mil imagens para teste. Exemplo do conjunto pode ser visto a seguir:



Este roteiro foi construído pensando em uma evolução natural a partir de Redes Neurais para Redes Convolucionais, tratando da construção da estrutura do modelo, o processo de ajuste e as avalições pertinentes.

**Redes Neurais - Introdução**

from tensorflow.keras.models import Sequential  
from tensorflow.keras.layers import Dense, Input  
from keras import ops

model = Sequential()

A construção do modelo pode ser passando uma lista de *layers* já na construção, ou adicionar utilizando o método *add()*. As camadas, do tipo *Dense*, são camadas densamente conectadas de uma Rede Neural (NN).

Na construção:

model = Sequential(

[

Dense(4, activation='relu', name='layers'),

Dense(2),

]

)

Adicionando posteriormente:

model = Sequential()

model.add(Dense(4, activation='relu'))

model.add(Dense(2)) #

Para verificação, pode-se utilizar:

print(model.summary())

O Dense representa a operação efetuada por uma camada de uma rede neural, no formato:

Portanto, fica claro aqui que cada *Dense layer* é a conexão da entrada com a próxima camada, ou a saída da camada anterior com a próxima camada. No exemplo, são 4 neurônios na camada de entrada e 2 neurônios na camada de saída.

Verifique os pesos, utilizando model.weights

x = ops.ones((1,4)) # retorna um tensor de dimensão 1 e 4. Ajuste a dimensão!

y = model(x)

Verifique agora os pesos e o resumo do modelo. Adicione *use\_bias=True* nas camadas e veja o que muda na sápida do modelo.

model.weights  
model.summary()

Podemos já, de antemão, iniciar a dimensão de entrada para ter as informações de parâmetros. Para isso, a primeira camada deve ser uma *Input()*:

Entrada

Neurônios

Neurônios

bias

bias

Input(shape=(1,4)) # vetor

Input(shape=(256,256,3)) # matrix 256

Faça a alteração e verifique o resumo do modelo.

Na sequência, o conceito de Deep Learning pede a inclusão das camadas *convolucionais* e camadas de *pooling*. Para inclusão destas camadas, adicionamos com o método add(), utilizando as novas camadas, Conv2D e MaxPooling2D. Importante: para essas opções, a entrada deve ser uma matriz com linhas, colunas e profundidade.

model = Sequential()

model.add(Input(shape=(250, 250, 3)))

model.add(Conv2D(32, 5, strides=2, activation='relu'))

model.add(Conv2D(32, 3, activation='relu'))

model.add(MaxPooling2D(strides=3))

model.summary()

Vamos analisar:

São 32 filtros, cada um com dimensão 5x5, stride 2, sem padding.

São 32 filtros, cada um com dimensão 3x3, stride 1, sem padding.

MaxPooling com dimensão 3x3, operação Max (valor máximo dentro da máscara 3x3)

Seguindo...

model.add(Conv2D(32, 3, activation='relu'))

model.add(Conv2D(32, 3, activation='relu'))

model.add(MaxPooling2D(strides=3))

model.add(Conv2D(32, 3, activation='relu'))

model.add(Conv2D(32, 3, activation='relu'))

model.add(MaxPooling2D(strides=2))

Qual a diferença ente essas duas linhas?

model.add(GlobalMaxPool2D())

model.add(Flatten())

model.add(Dense(10))

**Treinamento utilizando TensorFlow**

Para treinamento de um modelo criado, temos que configurar o treinamento e fazer o ajuste do modelo, através de, respectivamente:

model.compile()

model.fit()

Com o compile() podemos configurar o otimizador, a função de perca (loss) e a métrica que será usada para avaliar o treinamento. Com o fit(), os dados são quebrados em “batches” e usa estes dados por épocas de treinamento para atualização dos pesos, dados os conjuntos de treinamento e validação, se disponível.

Exemplo com NN:

import torch

import tensorflow as tf

import os

import numpy as np

import keras

from keras import layers

from keras import ops

inputs = keras.Input(shape=(784,), name='digits')

Qual seu entendimento sobre este conjunto de linhas?

x = layer.Dense(64, activation='relu', name='dense\_1')(inputs)

x = layer.Dense(64, activation='relu', name='dense\_2')(x)

output = layers.Dense(10, activation='softmax', name='predictions')(x)

model = keras.Model(inputs=inputs, outputs=outputs)

(x\_train, y\_train), (x\_test, y\_test) = keras.datasets.mnist.load\_data()

x\_train = x\_train.reshape(60000, 784).astype('float32') / 255

x\_test = x\_test.reshape(10000, 784).astype('float32') / 255

y\_train = y\_train.astype('float32')

y\_test = y\_test.astype('float32')

x\_val = x\_train[-10000:]

y\_val = y\_train[-10000:]

x\_train = x\_train[:-10000]

y\_train = y\_train[:-10000]

model.compile(

optimizer=keras.optimizers.RMSprop(),

loss=keras.lesses.SparseCategoricalCrossentropy(),

metrics=[keras.metrics.SparseCategoricalAccuracy()],  
)

history = model.fit(

x\_train,

y\_train,

batch\_size=64,

epochs=2,

validation\_data=(x\_val, y\_val),

)

print(history.history)

results = model.evaluate(x\_test, y\_test, batch\_size=128)

predictions = model.predict(x\_test[:3]

print('test loss, test acc: ', results, ' | predictions shape: ', predictions.shape)

**Exemplo com CNN:**

Para utilização da CNN, não há a necessidade de fazer o reshape da imagem para torna-la um vetor, uma vez que ela é utilizada como matriz na entrada. Aqui vamos utilizar as camadas Conv2D e MaxPooling2D até a saída. Além disso, configuração e ajuste (fit) continua o mesmo.

from tensorflow.keras.datasets import mnist

from tensorflow.keras.models import Sequential

from tensorflow.keras.layers import Conv2D, MaxPool2D, Flateen, Dropout, Dense

(x\_train, y\_train), (x\_test, y\_test) = mnist.load\_data()

x\_train = x\_train.reshape((x\_train.shape[0], x\_train.shape[1], x\_train.shape[2], 1))

x\_test = x\_test.reshape((x\_test.shape[0], x\_test.shape[1], x\_test.shape[2], 1))

print(x\_train.shape)

print(x\_test.shape)

x\_train = x\_train/255

x\_test = x\_test/255

model = Sequential()

model.add(Conv2D(32, 3, activation='relu', input\_shape=(28,28,1)))

model.add(MaxPool2D(2,2))

model.add(Flatten())

model.add(Dense(100, activation='relu'))

model.add(Dense(10, activation='softmax'))

model.compile(

loss='sparse\_categorical\_crossentropy',

optimizer='adam',

metrics=['accuracy'],

)

processo = model.fit(x\_train, y\_train, epochs=10)

model.evaluate(x\_test, y\_test)

model.predict(x\_test[:1])

plt.imshow(x\_test[:1])

Uma análise importante a ser fazer é verificar a evolução da curva de *loss* (função de perca), para o treinamento e para a validação. Esta análise pode nos dar informação valiosa sobre *overfitting* do modelo.

fig = plt.figure(figsize=(10,8))

ax = fig.gca()

ax.set(title='training X validation')

ax.plot(process.history['loss'], color='black', label='training\_loss')

ax.plot(process.history['val\_loss'], color='red', label='validation\_loss')

ax.set\_xlabel('Epochs')

ax.set\_ylabel('Loss')

ax.legend()

plt.show()

Outro exemplo, usando CNN para imagens em outro conjunto de dados bastante famoso, o cifar10. Vamos ao código:

import numpy as np

import matplotlib.pyplot as plt

from tensorflow.keras.utils import to\_categorical

from tensorflow.keras.models import Sequential

from tensorflow.keras.layers import Dense, Conv2D, MaxPool2D, Dropout, Flatten

from tensorflow.keras.datasets import cifar10

(x\_train, y\_train), (x\_test, y\_test) = cifar10.load\_data()

print(x\_train.shape)

print(y\_train.shape)

print(x\_test.shape)

print(y\_test.shape)

np.unique(y\_train)

img = 100

plt.subplot(121)

plt.imshow(x\_train[img])

plt.title('Label: {}'.format(y\_train[img]))

plt.subplot(122)

plt.imshow(x\_test[img])

plt.title('Label: {}'.format(y\_test[img]))

x\_train = x\_train / 255

x\_test = x\_test / 255

y\_train = y\_train.reshape(-1,)

y\_test = y\_test.reshape(-1,)

model=Sequential()

model.add(Conv2D(32, (3,3), activation='relu', input\_shape=(32,32,3)))

model.add(MaxPool2D(2,2))

model.add(Conv2D(64, (3,3), activation='relu'))

model.add(MaxPool2D(2,2))

model.add(Flatten())

model.add(Dense(216, activation='relu'))

model.add(Dense(10, activation='softmax'))

model.compile(optimizer='rmsprop',

loss='sparse\_categorical\_crossentropy',

metrics=['accuracy'],)

progresso = model.fit(x\_train, y\_train, epochs=10, validation\_split=0.2)

model.evaluate(x\_test,y\_test)

plt.plot(progresso.history['loss'], color='red')

plt.plot(progresso.history['val\_loss'], color='black')

plt.show()

pred = model.predict(x\_test)

print(pred[0])

print('Index:', np.argmax(pred[0]))

y\_classes = [np.argmax(element) for element in pred]

print('Predicted\_values: ', y\_classes[:10])

print('Actual\_values: ', y\_test[:10])

Por último, analisando os resultados é possível notar que o treinamento não foi tão satisfatório. O modelo teve um *overtraining* e podemos tomar algumas ações para evitar esse efeito. Uma das formas de fazer é inserir uma camada em alguns pontos do modelo, chamada de camada de *Dropout*.

Essa é uma técnica de regularização em que alguns elementos são levados a zero (*drop out*) durante o treinamento. A ideia é não reforçar tanto algumas conexões evitando o *overtraining*. A seguir, inclua essas novas camadas conforme a descrição e faça um novo treinamento do modelo, avaliando os resultados obtidos.

1. Adicione uma camada Dropout(fator) após as camadas MaxPool2D e Dense. Refaça o treinamento, a avaliação, analise os gráficos das curvas treinamento x validação e as predições feitas.

Importante: o fator na camada Dropout está relacionada com a frequência em que a uma unidade é levada a 0, em cada etapa do processo de treinamento. O valor vai de 0.0 a 1.0.

1. Se necessário, refaça a análise com outros valores para fator.