KERAS

João Pedro Mazuco Rodriguez

<https://keras.io/>

- Sobre:

O Keras é uma API em Python, que dispõe de funções para a construção de redes neurais convolucionais (CNN), rodando o framework Tensorflow, da Google, ou o Theanos, ou ainda o CNTK, da Microsoft, no backend. O Keras vende-se por possuir um código mais fluído que o tensorflow e obter o mesmo desempenho. Pelo fato do Tensorflow ser escrito em C++, mesmo sua versão em Python possui um desenvolvimento parecido com C++. O Keras portanto, vende-se como o “Tensorflow com cara de Python”. São APIs que permitem uma implementação e um aprendizado muito mais fácil.

Além de funções prontas para criação de uma camada de neurônios densamente conectadas, por exemplo, com um simples Dense(), o Keras dispõe de diversas funções de ativação, backpropagation e de saves de modelos. Além disso, a vida do programador é muito facilitada tanto para o aprendizado, com datasets prontos para serem usados para treinos e validação, quanto funções para criar seu dataset e podendo aplicar diversas transformações de imagens.

- Instalação

Pelo fato de ser um dos frameworks mais utilizados e famosos para Deep Learning, e por pertencer na maior empresa de Big Data do mundo, a Google, utilizarei no trabalho o Tensorflow no backend, mesmo sendo possível utilizar outros.

A parte que exige um pouco mais para a instalação é o Tensorflow. Por ter para Windows, Linux e macOS, não irei me ater muito a cada um dos OS, irei apenas demonstrar no Windows, que é o OS mais utilizado, apesar de não ser o mais recomendado, principalmente pela questão de desempenho.

Link de instalação Tensorflow para todos os OS: <https://www.tensorflow.org/install/>

Para Windows, é necessário a instalação do Python 3.5 ou 3.6 (<https://www.python.org/downloads/release/python-362/>). O Python 3 já fornece no comando pip3 a instalação do tensorflow com o comando:

C:\> **pip install tensorflow**

Feito a instalação do Tensorflow, o keras é instalado apenas com: (se estiver no anaconda faça isso dentro do ambiente do tensorflow: activate tensorflow)

C:> **pip install keras**

Para poder utilizar o código square.py, instale também o pillow:

C:> **pip install pillow**

- Códigos

Com tudo instalado, vamos aos códigos. Primeiramente irei disponibilizar algumas vídeo-aulas que podem ser úteis para quem quiser aprofundar-se:

Redes Neurais e Keras: <https://www.youtube.com/watch?v=KIvB5LFbA0w>

Como funcionam as convoluções (neurônios que possuem não só um peso, mas uma matriz deles): <https://www.youtube.com/watch?v=DXnyuUZcAAI>

Iremos, no primeiro programa, realizar uma rede neural simples com 3 camadas densamente conectadas, para classificar dígitos de 0 a 9, usando o dataset pronto MNIST:



Imagem 1. Dataset MNIST

# Importamos o conjunto de dados MNIST

from keras.datasets import mnist

# Para visualizarmos bem a sequencia de camadas do modelo

# vamos usar o modulo do Keras chamado Sequential

from keras.models import Sequential

# O Dense irá possibilitar camadas de neurônios densamente

# conectadas

from keras.layers import Dense

# Modulo do Keras responsavel por varias rotinas de

# pre-processamento

from keras.utils import np\_utils

# Aqui carregamos os dados do dataset pronto MNIST para vetores

# O vetor X\_train possui os dados de entrada, o y\_train suas

# saídas classificadas. O X\_test e y\_test a mesma coisa porém

# para validacao

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

# Nestas linhas estamos transformando nossas matrizes de entrada  
# (imagens) em vetores, para conseguir ser passados como

# parâmetros

num\_pixels = X\_train.shape[1] \* X\_train.shape[2]

X\_train = X\_train.reshape(X\_train.shape[0],num\_pixels).astype('float32')

X\_test = X\_test.reshape(X\_test.shape[0],num\_pixels).astype('float32')

# Aqui estamos apenas transformando valores de grayscale(0 a

# 255) em um intervalo de 0 a 1. Isto ajuda nas multiplicações

# principalmente por problemas causados computacionalmente em

# multiplicações de números muito grandes por números muito

# pequenos ou vice-versa

X\_train = X\_train / 255

X\_test = X\_test / 255

# Como estamos trabalhando com um problema de classificacao

# multiclasses, pois temos varios tipos de digitos, vamos

# representa-los em categorias usando a metodologia de

# one-hot-encoding aqui representada pela funcao to\_categorical.

y\_train = np\_utils.to\_categorical(y\_train)

y\_test = np\_utils.to\_categorical(y\_test)

# Numero de saídas possíveis (0 a 9) no caso do mnist.

num\_classes = y\_test.shape[1]

#Rede Neural propriamente dita:

def base\_model():

model = Sequential()

model.add(Dense(num\_pixels, input\_dim=num\_pixels, activation='relu'))

model.add(Dense(64, input\_dim=num\_pixels, activation="relu"));

model.add(Dense(num\_classes, activation='softmax', name='preds'))

model.compile(loss='categorical\_crossentropy', optimizer='sgd', metrics=['accuracy'])

return model

# chamo a rede neural e guardo seu modelo na variável

# model

model = base\_model()

# O metodo summary revela quais sao as camadas

# que formam o modelo, seus formatos e o numero

# de parametros envolvidos em cada etapa.

model.summary()

# Processo de treinamento do modelo.

model.fit(X\_train, y\_train, validation\_data=(X\_test, y\_test), epochs=1, batch\_size=100)

Entrando em detalhes agora no modelo criado: Ao usarmos as funções model.add(Dense()), estamos criando uma camada de neurônios densamente conectada, sequencial (por isso a função Sequential() no início). Os parâmetros passados, são, o número de neurônios (784, 64 e 10, respectivamente), a dimensão da entrada (28x28) e a função de ativação (aqui a RELU, que funciona basicamente com a imagem abaixo):

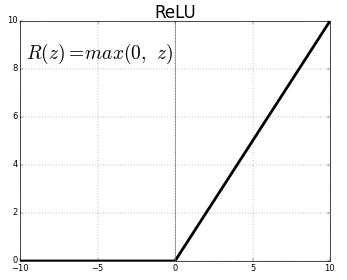


Imagem 2. Função RELU (se x<0, x=0, senão x=x)

Após a criação das camadas, o modelo será treinado com o backpropagation, definido na função model.compile(), com os parâmetros loss que será sempre categorical\_crossentropy quando haver mais do que duas saídas nos neurônios ou binary\_crossentropy quando houver duas saídas, optmizer que será a função de otimização para trocar os pesos e bias dos neurônios (no caso estamos usando sgd, mas há diversos) e metrics que irá mostrar os valores em taxa de acerto durante a execução.

Por fim realizamos o model.summary() que irá mostrar os parâmetros e as camadas no prompt e o model.fit() que finalmente, será a função que treinará nosso modelo. Os parâmetros são os vetores de treino e validação, epochs, que são o número de treinos, e batch\_size, que será a execução em lotes: a cada x vezes (numero do parâmetro) é realizado a mudança dos pesos.

Realizamos aqui um modelo simples de rede neural, que, com apenas 3 camadas, realizando troca de pesos de 100 em 100 entradas, acerta mais que 90% das entradas. Sinta-se livre para adicionar novas camadas, trocar o número de neurônios, batch\_size, epochs e até otimizador (teste o adam, por exemplo!). Lembre-se que se estiver utilizando CPU, o desempenho cai exponencialmente se colocar muitas camadas com muitos neurônios.

Iremos aumentar a taxa de acerto significativamente no próximo código (Usaremos agora convoluções, isto é, neurônios com matrizes de pesos, com a função Conv2D). Convolução é uma técnica muito utilizada em Deep Learning, pois ao se ter muitos dados e uma rede enorme, redes neurais simples começam a se tornar muito pesadas e lentas. As convoluções detectam padrões em 3x3 pixels, por exemplo, e não em apenas 1, realizando detecção de padrões em vários pixels de uma vez.

from keras.datasets import mnist

from keras.models import Sequential

from keras.layers import Dense

from keras.layers import Dropout

from keras.layers import Flatten

import numpy as np

from keras.layers.convolutional import Conv2D

from keras.layers.convolutional import MaxPooling2D

from keras.utils import np\_utils

from keras import backend as K

K.set\_image\_dim\_ordering('th')

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

X\_train = X\_train.reshape(X\_train.shape[0], 1, 28, 28).astype('float32')

X\_test = X\_test.reshape(X\_test.shape[0], 1, 28, 28).astype('float32')

X\_train = X\_train / 255

X\_test = X\_test / 255

y\_train = np\_utils.to\_categorical(y\_train)

y\_test = np\_utils.to\_categorical(y\_test)

def deeper\_cnn\_model():

model = Sequential()

model.add(Conv2D(60, (5, 5), input\_shape=(1, 28, 28), activation='relu'))

model.add(MaxPooling2D(pool\_size=(2, 2)))

model.add(Conv2D(30, (3, 3), activation='relu'))

model.add(Conv2D(30, (3, 3), activation='relu'))

model.add(Dropout(0.2))

model.add(Flatten())

model.add(Dense(128, activation='relu'))

model.add(Dense(64, activation='relu'))

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

model.compile(loss='categorical\_crossentropy', optimizer='adam', metrics=['accuracy'])

return model

model = deeper\_cnn\_model()

model.summary()

model.fit(X\_train, y\_train, validation\_data=(X\_test, y\_test), epochs=5, batch\_size=100)

scores = model.evaluate(X\_test, y\_test, verbose=0)

print("Erro de: %.2f%%" % (100-scores[1]\*100))

O código é praticamente, o mesmo, o que mudou foi a nossa função model, que agora possui matrizes de convolução e outras técnicas. Para criar um neurônio “matriz” convolucional é utilizado a função Conv2D(), que possui como parâmetros o número de neurônios e o tamanho da matriz, além da função de ativação. A função Dropout é utilizada para não criar overfitting, isto é, para não “viciar” a rede. Normalmente, quando é treinado muitas vezes com um mesmo dataset, o computador começa a deixar de generalizar. Ele possui uma taxa de acerto muito alta no treino, mas na hora de validar sua taxa é muito menor.

Para contornar isto o Dropout surge como uma alternativa. Basicamente esta função “desativa” aleatoriamente x% dos neurônios da rede (no caso do nosso código, 20% (0.2)) fazendo com que se treine parte da rede de cada vez, e na hora de validar utilize ela em seu 100%. Esta técnica permite com que o overfitting seja minimizado e é muito utilizado.

A função MaxPooling2D basicamente pega o maior pixel dentre 4 e transforma em apenas 1. Isto é, uma matriz: [2, 4][8,2] irá virar apenas 8. Essa técnica é usada para diminuir o número de pixels de uma imagem, diminuindo sua resolução. É muito utilizada quando treinamos com imagens muito grandes, fazendo com que possamos fazer uma rede neural menor e mais leve com uma capacidade equivalente de acerto.

Aqui utilizei a função evaluate(), ela serve basicamente para usar o modelo treinado para avaliar sua taxa de acerto. Ele retorna essa taxa num valor entre 0 e 1. Geralmente é utilizada para testar a rede com um dataset novo, que não teve participação nos treinos. Aqui utilizei o próprio dataset de validação.

Basicamente esta foi a biblioteca Keras, utilizando os datasets prontos da biblioteca. Esses datasets vem apenas com o número dos pixels em rgb, o que não é o caso de arquivos de imagem, que possuem cabeçalhos e técnicas de compactação. Para isso, vou mostrar a função utilizada para usar um dataset próprio de imagens PNG, BMP, JPG, PPM ou TIF. Essas funções transformam uma pasta de arquivos de imagens em um vetor único de treino e as próprias pastas viram um critério de saída, isto é, para fazer um modelo que reconheça cadeiras e sofás, por exemplo, farei duas pastas (de treino e de validação, apenas para organizar) e para cada uma delas separarei uma pasta com imagens de cadeiras e outras de sofás. A classe ImageDataGenerator já possui funções que transformará tudo nos vetores para o treino e funções que permitem alterar as propriedades da imagem, rotacionando, escalando, etc, como veremos a seguir:

## Deixarei aqui apenas os códigos das funções que treinarão e puxarão

## as imagens do diretório. Para criar os modelos, tudo funciona como  
## mostrei nos códigos anteriores.

train\_datagen = ImageDataGenerator(

rescale=1./255,

zoom\_range=0.2,

horizontal\_flip=True,

vertical\_flip=True,

rotation\_range=90)

test\_datagen = ImageDataGenerator(rescale=1./255)

X\_train = train\_datagen.flow\_from\_directory(

'data/dataset',

target\_size=(120, 120),

batch\_size=300,

color\_mode='grayscale')

y\_train = train\_datagen.flow\_from\_directory(

'data/validation',

target\_size=(120, 120),

batch\_size=40,

color\_mode='grayscale')

model.fit\_generator(

X\_train,

steps\_per\_epoch=500,

epochs=epochs,

validation\_data=y\_train,

validation\_steps=50)

O construtor da classe ImageDataGenerator possui os parâmetros de redimensionamento das imagens. Aqui estou transformando os valores de 0 a 255 em 0 a 1, dando um zoom aleatório de até 20% nas imagens, fazendo flips horizontais e verticais e rotações de 90 graus. Guardo na variável train\_datagen. Na test\_datagen (que será meu dataset de validação) apenas troco os valores.   
 A função flow\_from\_directory será a que vai puxar os valores do diretório. O primeiro parâmetro é o path das imagens (repare que não é necessário fazer de um a um para cada classe (cadeiras e sofás, por exemplo), pois ele já o faz automaticamente). Os outros parâmetros são de redimensionamento da imagem (é necessário que tenham todas o mesmo tamanho, se não tiver use este parâmetro), o tamanho do batch (funcionando da mesma forma que o batch\_size explicado no primeiro exemplo) e o cor, que era rgb e eu estou transformando em grayscale. Existem o parâmetro

O model.fig\_generator será o treinamento para este tipo de entrada, em que é passado somente o vetor de treino gerado, o numero do batch, o número de epochs, o vetor de validação gerado e o batch de validação.