*Reconhecimento De Máscaras Utilizando Métodos De Detecção De Objetos Baseados Em Redes Neurais Convolucionais*

Márcio João Ferreira

**DETALHES DA IMPLEMENTAÇÕES DOS FRAMEWORKS API OBJECT DETECTION E YOLO**

Sumário

[Estrutura de diretórios API Tensorflow Object Detection 3](#_Toc139036088)

[Estrutura de diretórios YOLO 7](#_Toc139036089)

[Exemplo de padrão de anotações Pascal VOC e YOLO 10](#_Toc139036090)

[Script de Índices das Imagens – YOLO 11](#_Toc139036091)

[Script dos arquivos .record - Tensorflow API Object Detection 12](#_Toc139036092)

[Script de Instalação Tensorflow API Object Detection 13](#_Toc139036093)

[Script de Instalação Darknet/YOLO 14](#_Toc139036094)

[Script para Inferência - Tensorflow API Object Detection 15](#_Toc139036095)

[Script para Inferência - Darknet 16](#_Toc139036096)

# Estrutura de diretórios API Tensorflow Object Detection

A instalação e configuração pode ser realizada através do tutorial oficial da API. Seguindo o tutorial, a preparação do ambiente para o treinamento com Faster R-CNN e SSD ficou organizado da seguinte forma no Google Drive: um diretório raiz chamado “*tensorflow*” e dois subdiretórios “*scripts*” e “*workspace*”.

No diretório “*scripts*” temos um subdiretório “*preprocessing*”, que contém o *script generate\_tfrecord.py* para geração dos arquivos *train.record* e *test.record*. Os *scripts* estão descritos no Apêndice 5 deste trabalho. Esses dois arquivos são obrigatórios para o funcionamento da API, eles contêm os índices dos locais das imagens para o treinamento e teste. No terminal, o comando utilizado para gerar os arquivos *train.record* e *test.record*, respectivamente são:

* *Python generate\_tfrecord.py -x [“caminho das imagens de treino”] /train -l [“caminho do diretório annotations”]/label\_map.pbtxt -o [“caminho do diretório annotations”]/train.record*;
* *Python generate\_tfrecord.py -x [“caminho das imagens de teste”]/test -l [“caminho do diretório annotations”]/label\_map.pbtxt -o [“caminho do diretório annotations”]/test.record*.

O diretório “*workspace*” contém um subdiretório nomeado como “*trained\_demo*”. No diretório “*trained\_demo*” temos os subdiretórios “*annotations*”, “*exported-models*”, “*images*”, “*models*” e “*pre-trained-models*”, o papel de cada um deles são descritos da seguinte maneira:

* *Annotations* – Contém um arquivo *label\_map.pbtxt* (arquivo criado em branco com o aplicativo Notepad++), que contém o nome das categorias. No mesmo local, ainda se encontra os arquivos *train.record* e *test.record*;
* *Exported-models* – Local onde será salvo o modelo para inferência;
* *Images* – Contém subdiretórios “*train*” e “*test*”, respectivamente as imagens de treino e as imagens de teste;
* *Models* – local onde armazenam os pesos treinados com o arquivo de índice “ckpt-1.index”, “ckpt-2.index” e assim por diante. Nesse local, temos um arquivo de *checkpoint* (usado para continuar o treinamento de um ponto de parada) e um arquivo de configuração *pipeline.config*, onde os parâmetros alterados foram:
  1. *num\_classes*: número de classes a ser detectado;
  2. *batch\_size*: tamanho do lote;
  3. *fine\_tune\_checkpoint*: local dos pesos pré-treinado;
  4. *fine\_tune\_checkpoint\_type*: “*detection*”;
  5. *use\_bfloat16*: false;
  6. *label\_map\_path*: local do arquivo *label\_map.pbtxt*;
  7. *train\_input\_reader e input\_path*: local do arquivos *train.record*;
  8. *metrics\_set*: Alterado para "*pascal\_voc\_detection\_metrics* ";
  9. *eval\_input\_reader e input\_path*: local do arquivo *test.record*;
  10. *num\_steps*: total de iterações.
* *Pre-trained-models*: contém o arquivo *pipeline.config* original e um subdiretório com o nome “*checkpoint*” com o *download* dos pesos do modelo pré-treinado baixado do repositório.

As Figuras 1 e 2 são exemplos de configurações utilizadas nos arquivos *label\_map.pbtxt* e *pipeline.config*. No exemplo foi utilizado modelo pré-treinado SSD Resnet-50 v1 FPN.

Interface gráfica do usuário, Texto, Aplicativo, Email

Descrição gerada automaticamente

Figura 1 – Parâmetros do arquivo *label\_map.pbtxt*.

Texto

Descrição gerada automaticamente com confiança baixa

Figura 2 – Parâmetros do arquivo *pipeline.config*.

**Execução do treinamento da *Tensorflow API Object Detection***

Para facilitar a execução dos comandos de treinamento, exportação do modelo e avaliação, dois arquivos “*model\_main\_tf2.py*” (*script* de treinamento) e “*exporter\_main\_v2.py*” (*script* de exportação do modelo) foram copiados do diretório de instalação da *Tensorflow API Object Detection* para o diretório “*trained\_demo*”. Considerando uma nova célula do terminal, no nível do diretório “*trained\_demo*”, usando os pesos da arquitetura SSD Resnet 50 v1 FPN, os comandos são mostrados conforme Tabela 1.

|  |  |
| --- | --- |
| **Descrição** | **Comando no terminal** |
| Treinamento | *python model\_main\_tf2.py --model\_dir=models/1\_my\_ssd\_resnet50\_v1\_fpn --pipeline\_config\_path=models/1\_my\_ssd\_resnet50\_v1\_fpn/pipeline.config* |
| Exportação do modelo | *python ./exporter\_main\_v2.py --input\_type image\_tensor --pipeline\_config\_path ./models/1\_my\_ssd\_resnet50\_v1\_fpn/pipeline.config --trained\_checkpoint\_dir ./models/1\_my\_ssd\_resnet50\_v1\_fpn/ --output\_directory ./exported-models/1\_my\_ssd\_resnet50\_v1\_fpn* |
| Avaliação | *python model\_main\_tf2.py --model\_dir=models/1\_my\_ssd\_resnet50\_v1\_fpn --pipeline\_config\_path=models/1\_my\_ssd\_resnet50\_v1\_fpn/pipeline.config --checkpoint\_dir=models/1\_my\_ssd\_resnet50\_v1\_fpn* |

Tabela 1 – Comandos utilizados para *Tensorflow API Object Detection*.

Os parâmetros dos s*cripts model\_main\_tf2.py* e *exporter\_main\_v2.py*, são descritos a seguir:

* *--model\_dir*; *--trained\_checkpoint\_dir* e *--checkpoint\_dir*: representa o local onde serão armazenados sos pontos de parada dos pesos. Para cada modelo foi criado um diretório com o nome da arquitetura utilizada;
* --pipeline\_config\_path: local do arquivo *pipeline.config*;
* --output\_directory: local onde será salvo o modelo para inferência;

# Estrutura de diretórios YOLO

A preparação do ambiente para o treinamento das arquiteturas YOLO ficou organizado da seguinte forma no Google Drive: um diretório raiz nomeado como “yolo”em um subdiretório chamado “recursos”, como segue:

* Em “recursos” contém dois *scripts* *gerar\_train.py* e *gerar\_test.py*, que geram respectivamente os arquivos de índices das imagens *train.txt* e *test.txt*. Os scripts estão descritos no Apêndice 4 deste trabalho.
* No diretório “recursos” foi criado um subdiretório com o nome “*data*”, que contém os subdiretórios “*train*” e “*test*” com a divisão das imagens de treino e teste, respectivamente. Ainda no diretório “*data*” se encontra os arquivos *train.txt* e *test.txt*.

O diretório “recursos” é o local onde armazenam os pesos treinados com a extensão .*weights*, por exemplo, o arquivo “*yolov4\_custom\_best.weights*”. Além disso, está presente nesse mesmo local, os arquivos *yoloxx\_custom.cfg*, *obj.data* e *obj.names*. As características desses arquivos são descritas da seguinte maneira:

* *yoloxx\_custom –* onde *xx* foi utilizado para indicar a versão v2, v3 ou v4 (por exemplo: *yolov4\_custom*). Esse arquivo foi criado a partir de uma cópia do arquivo *yoloxx.cfg*, que está dentro do diretório “cgf” do repositório *Darknet*. Os parâmetros alterados foram:
  1. *batch*: tamanho do lote.
  2. *subdivisions*: subdivisão do *batch*;
  3. *max\_batches*: total de iterações.
  4. *steps*: o ponto de verificação onde as escalas serão aplicadas. Valor sugerido na documentação é 80% / 90% do *max\_batches*;
  5. *filters*: filtro da camada convolucional, que antecede a camada de detecção. Valor sugerido pela documentação (5 + 2 classes)×3;
  6. *classes*: número de classes a ser detectado;
* *obj.data –* arquivo em branco criado para adicionar os seguintes parâmetros:
  1. *classes*: número de classes a ser detectado;
  2. *train*: local das imagens de treino;
  3. *valid*: local das imagens de teste;
  4. *names*: local do arquivo obj.names;
  5. *backup*: local onde será salvo o modelo para inferência.
* *obj.names –* arquivo em branco criado para adicionar os nomes das categorias.

As figuras 3, 4 e 5 são exemplos de configurações utilizadas nos arquivos *yoloxx\_custom.cfg*, *obj.data* e *obj.names*, que auxiliam o *framework* *Darknet*/YOLO. No exemplo, foi utilizado o modelo pré-treinado YOLOv4.

Uma imagem contendo Linha do tempo

Descrição gerada automaticamente

Figura 3 – Parâmetros do arquivo *yoloxx\_custom.cfg*.

Interface gráfica do usuário, Texto

Descrição gerada automaticamente

Figura 4 – Parâmetros do arquivo *obj.data*.

Interface gráfica do usuário, Texto, Aplicativo, chat ou mensagem de texto

Descrição gerada automaticamente

Figura 5 – Parâmetros do arquivo *obj.names*.

Para trabalhar com GPU no *Darknet*, é necessário a alteração de algumas linhas no arquivo MakeFile localizado no diretório raiz do repositório *Darknet*, alterar para o valor 1 os parâmetros abaixo:

* OPENCV=1;
* GPU=1;
* CUDNN=1.

**Execução do treinamento *Darknet*/YOLO**

No *framework* *Darknet* o comando “*detector*” é utilizado para realização do treinamento e da avaliação. Considerando uma nova célula do terminal, no nível do diretório “*Darknet*”, após os procedimentos de instalação descrito na Apêndice 7 (instalação *Darknet*/YOLO), usando os pesos do YOLOv4, a descrição e os comandos são mostrados conforme Tabela 12.

|  |  |
| --- | --- |
| **Descrição** | **Comando no terminal** |
| Treinamento | *./darknet detector train data/obj.data cfg/yolov4\_custom.cfg  yolov4.conv.137 -dont\_show -map* |
| Avaliação | *./darknet detector map data/obj.data cfg/yolov4\_custom.cfg /yolo/recursos/yolov4\_custom\_final.weights* |

Tabela 2 – Comandos utilizados para *Darknet*/YOLO.

Os parâmetros do comando “*detector*” do *framework* *Darknet*, são descritos a seguir:

* *train* – usado para informar que o comando “*detector*”será utilizado para iniciar um treinamento, junto do parâmetro *-map* indica que será fornecido um gráfico durante o processo de treinamento mostrando o desempenho da rede neural;
* *map* – usado para informar que o comando “*detector*” será utilizado para avaliar o modelo;

Os demais parâmetros indicam os caminhos dos arquivos que auxiliam o *framework*, os locais de armazenamento dos modelos pré-treinados e do modelo final obtido.

# Exemplo de padrão de anotações Pascal VOC e YOLO

**Pascal VOC**

As informações contidas no XML do padrão Pascal VOC são: os pontos *x\_min, y\_min, x\_max, y\_max* e o nome da categoria objeto.

Interface gráfica do usuário, Texto, Aplicativo, Email

Descrição gerada automaticamente

Anotação Padrão Pascal VOC.

**YOLO**

Cada linha representa um objeto com as informações. As informações contidas no TXT do padrão YOLO são: Id da categoria, coordenada x, coordenada y, largura e altura, que são obtidos da seguinte forma:

* Largura: largura do objeto dividido pela largura da imagem;
* Altura: altura do objeto dividido pela altura da imagem;
* X e Y: são pontos centrais da delimitação da caixa;
* Id da categoria.

Texto

Descrição gerada automaticamente

Anotação padrão YOLO.

# Script de Índices das Imagens – YOLO

O*s* s*cripts*“gerar\_train.py” e “gerar\_test.py”, geram respectivamente os arquivos train.txt e test.txt. De acordo com a organização dos diretórios na seção 5.4, os *scripts* devem ser executados no nível do diretório “./recursos”.

**Arquivo: gerar\_train.py**

#O que o código abaixo faz: adiciona o nome e caminho num arquivo txt chamado train.txt

**import os**

**imagens = []**

**os.chdir(os.path.join("data","train"))**

**forfilenameinos.listdir(os.getcwd()):**

**iffilename.endswith(".jpg"):**

**imagens.append("data/test/" + filename)**

**os.chdir("..")**

**with open("train.txt","w") as outfile:**

**for img in imagens:**

**outfile.write(img)**

**outfile.write("\n")**

**outfile.close()**

**os.chdir("..")**

**Arquivo: gerar\_test.py**

#O que o código abaixo faz: adiciona o nome e caminho num arquivo txt chamado test.txt

**import os**

**imagens = []**

**os.chdir(os.path.join("data","test"))**

**forfilenameinos.listdir(os.getcwd()):**

**iffilename.endswith(".jpg"):**

**imagens.append("data/test/" + filename)**

**os.chdir("..")**

**with open("test.txt","w") as outfile:**

**for img in imagens:**

**outfile.write(img)**

**outfile.write("\n")**

**outfile.close()**

**os.chdir("..")**

# Script dos arquivos .record - Tensorflow API Object Detection

O presente script foi nomeado como *generate\_tfrecord.py*, que serve para a criação dos arquivos train.record e test.record.

import os

importglob

import pandas aspd

importio

importxml.etree.ElementTreeasET

importargparse

os.environ['TF\_CPP\_MIN\_LOG\_LEVEL'] = '2' # SuppressTensorFlowlogging (1)

import tensorflow.compat.v1 as tf

from PIL importImage

fromobject\_detection.utilsimportdataset\_util, label\_map\_util

fromcollectionsimportnamedtuple

# Initiateargument parser

parser = argparse.ArgumentParser(

description="Sample TensorFlow XML-to-TFRecord converter")

parser.add\_argument("-x",

"--xml\_dir",

help="Path to the folder where the input .xml files are stored.",

type=str)

parser.add\_argument("-l",

"--labels\_path",

help="Path to the labels (.pbtxt) file.", type=str)

parser.add\_argument("-o",

"--output\_path",

help="Path of output TFRecord (.record) file.", type=str)

parser.add\_argument("-i",

"--image\_dir",

help="Path to the folder where the input image files are stored. "

"Defaults to the samedirectory as XML\_DIR.",

type=str, default=None)

parser.add\_argument("-c",

"--csv\_path",

help="Path of output .csv file. Ifnoneprovided, then no file willbe "

"written.",

type=str, default=None)

args = parser.parse\_args()

ifargs.image\_dirisNone:

args.image\_dir = args.xml\_dir

label\_map = label\_map\_util.load\_labelmap(args.labels\_path)

label\_map\_dict = label\_map\_util.get\_label\_map\_dict(label\_map)

defxml\_to\_csv(path):

xml\_list = []

for xml\_fileinglob.glob(path + '/\*.xml'):

tree = ET.parse(xml\_file)

root = tree.getroot()

filename = root.find('filename').text

width = int(root.find('size').find('width').text)

height =int(root.find('size').find('height').text)

formemberinroot.findall('object'):

bndbox = member.find('bndbox')

value = (filename,

width,

height,

member.find('name').text,

int(bndbox.find('xmin').text),

int(bndbox.find('ymin').text),

int(bndbox.find('xmax').text),

int(bndbox.find('ymax').text),

)

xml\_list.append(value)

column\_name = ['filename', 'width', 'height',

'class', 'xmin', 'ymin', 'xmax', 'ymax']

xml\_df = pd.DataFrame(xml\_list, columns=column\_name)

returnxml\_df

defclass\_text\_to\_int(row\_label):

returnlabel\_map\_dict[row\_label]

defsplit(df, group):

data = namedtuple('data', ['filename', 'object'])

gb = df.groupby(group)

return[data(filename, gb.get\_group(x)) forfilename, x in zip(gb.groups.keys(), gb.groups)]

defcreate\_tf\_example(group, path):

withtf.gfile.GFile(os.path.join(path, '{}'.format(group.filename)), 'rb') asfid:

encoded\_jpg = fid.read()

encoded\_jpg\_io = io.BytesIO(encoded\_jpg)

image = Image.open(encoded\_jpg\_io)

width, height = image.size

filename = group.filename.encode('utf8')

image\_format = b'jpg'

xmins = []

xmaxs = []

ymins = []

ymaxs = []

classes\_text = []

classes = []

for index, rowingroup.object.iterrows():

xmins.append(row['xmin'] / width)

xmaxs.append(row['xmax'] / width)

ymins.append(row['ymin'] / height)

ymaxs.append(row['ymax'] / height)

classes\_text.append(row['class'].encode('utf8'))

classes.append(class\_text\_to\_int(row['class']))

tf\_example = tf.train.Example(features=tf.train.Features(feature={

'image/height': dataset\_util.int64\_feature(height),

'image/width': dataset\_util.int64\_feature(width),

'image/filename': dataset\_util.bytes\_feature(filename),

'image/source\_id': dataset\_util.bytes\_feature(filename),

'image/encoded': dataset\_util.bytes\_feature(encoded\_jpg),

'image/format': dataset\_util.bytes\_feature(image\_format),

'image/object/bbox/xmin': dataset\_util.float\_list\_feature(xmins),

'image/object/bbox/xmax': dataset\_util.float\_list\_feature(xmaxs),

'image/object/bbox/ymin': dataset\_util.float\_list\_feature(ymins),

'image/object/bbox/ymax': dataset\_util.float\_list\_feature(ymaxs),

'image/object/class/text': dataset\_util.bytes\_list\_feature(classes\_text),

'image/object/class/label': dataset\_util.int64\_list\_feature(classes),

}))

returntf\_example

defmain(\_):

writer = tf.python\_io.TFRecordWriter(args.output\_path)

path = os.path.join(args.image\_dir)

examples = xml\_to\_csv(args.xml\_dir)

grouped = split(examples, 'filename')

for groupingrouped:

tf\_example = create\_tf\_example(group, path)

writer.write(tf\_example.SerializeToString())

writer.close()

print('Successfullycreated the TFRecord file: {}'.format(args.output\_path))

ifargs.csv\_pathisnotNone:

examples.to\_csv(args.csv\_path, index=None)

print('Successfullycreated the CSV file: {}'.format(args.csv\_path))

if \_\_name\_\_ == '\_\_main\_\_':

tf.app.run()

# Script de Instalação Tensorflow API Object Detection

O script mostra os passos para o treinamento dos métodos Faster R-CNN e SSD. O script parte da premissa que o terminal está na raiz Google Colaboratory“/content/”.

#Montagem do Google Drive onde contém a estrutura de diretórios e base de imagens.

from google.colab import drive

drive.mount('/content/drive')

#Download do *framework* da API Tensorflow Object Detection:

!git clone --depth 1 https://github.com/tensorflow/models

#Correção da métrica pascal voc metrics

%rm /content/models/*Research*/object\_detection/utils/object\_detection\_evaluation.py

%cp /content/drive/MyDrive/tensorflow/workspace/training\_demo/object\_detection\_evaluation.py /content/models/*Research*/object\_detection/utils

#A API usa o Protobufs para configurar e treinar parâmetros.  
%cd models/*Research*/

!protoc object\_detection/protos/\*.proto --python\_out=.

#No TensorFlow 2.x pycocotools é uma dependência da API Object Detection

!git clone https://github.com/coco*dataset*/cocoapi.git

%Cd cocoapi/PythonAPI

!make

cp -r pycocotools /content/models/*Research*/

%cd ..

%cd ..

#Instalação da API *Object Detection* 0.1

!cp object\_detection/packages/tf2/setup.py .

!python -m pip install .

Configuração para melhor desempenho da GPU.

#GPU

!pip freeze | grep Tensorflow

!pip uninstall tf-models-official –yes

!pip install tensorflow==2.4

!pip install tf-models-official==2.4

!pip freeze | grep Tensorflow

#Ponto de execução dos comandos para o treinamento, avaliação e exportação do modelo.

%cd /content/drive/MyDrive/tensorflow/workspace/training\_demo

#Para facilitar a execução do commando de treinamentofoi copiado do diretorio “/content/models/*Research*/object\_detection” os arquivos model\_main\_tf2.py e exporter\_main\_v2.py para o local “/content/drive/MyDrive/tensorflow/workspace/training\_demo”.

!pythonmodel\_main\_tf2.py--model\_dir=models/1\_my\_ssd\_mobilenet\_v2\_fpnlite\_320--pipeline\_config\_path=models/1\_my\_ssd\_mobilenet\_v2\_fpnlite\_320/pipeline.config

#Avaliação

!pythonmodel\_main\_tf2.py--model\_dir=models/1\_my\_ssd\_mobilenet\_v2\_fpnlite\_320--pipeline\_config\_path=models/1\_my\_ssd\_mobilenet\_v2\_fpnlite\_320/pipeline.config--checkpoint\_dir=models/1\_my\_ssd\_mobilenet\_v2\_fpnlite\_320

#Exportação do modelo para inferência

!python./exporter\_main\_v2.py--input\_typeimage\_tensor--pipeline\_config\_path./models/1\_my\_ssd\_mobilenet\_v2\_fpnlite\_320/pipeline.config--trained\_checkpoint\_dir./models/1\_my\_ssd\_mobilenet\_v2\_fpnlite\_320/--output\_directory./exported-models/1\_my\_ssd\_mobilenet\_v2\_fpnlite\_320

# Script de Instalação Darknet/YOLO

O script abaixo mostra os passos para o treinamento do método Yolo. O script parte da premissa que o terminal está na raiz /content/.

#Download do *framework* Darknet

!git clone https://github.com/AlexeyAB/darknet

#Alteração de parâmetros para trabalhar com GPU e modificação do caminho para geração do gráfico.

cd darknet/

!sed -i 's/OPENCV=0/OPENCV=1/' Makefile

!sed -i 's/GPU=0/GPU=1/' Makefile

!sed -i 's/CUDNN=0/CUDNN=1/' Makefile

!sed -i 's/chart.png/\/yolo\/recursos\/chart.png/g' src/image\_opencv.cpp

#Compilação do *framework*

!make

#Criação de link simbólico para facilitar a manipulação dos diretórios

!ln -s /content/drive/MyDrive/yolo /yolo

#Cópia dos arquivos de configuração

!cp /yolo/recursos/obj.data data/

!cp /yolo/recursos/obj.names data/

!cp /yolo/recursos/data/train.txt data/

!cp /yolo/recursos/data/test.txt data/

#Cópia da base de dados de treinamento e teste para o Google Colaboratory

!cp -r /yolo/recursos/data/train/ data/

!cp -r /yolo/recursos/data/test/ data/

#Download dos pesos pré-treinados

!wget https://github.com/AlexeyAB/darknet/releases/download/darknet\_yolo\_v3\_optimal/yolov4.conv.137

!wget https://pjreddie.com/media/files/darknet53.conv.74

!wget https://pjreddie.com/media/files/darknet19\_448.conv.23

#Cópia dos arquivos de configuração

!cp /yolo/recursos/yolov4\_custom.cfg cfg/

!cp /yolo/recursos/yolov3\_custom.cfg cfg/

!cp /yolo/recursos/yolov2\_custom.cfg cfg/

#Treinamento – Exemplo com o YOLOv4

!./darknet detector train data/obj.data cfg/yolov4\_custom.cfg  yolov4.conv.137 -dont\_show -map

#Avaliação

!./darknet detector map data/obj.data cfg/yolov4\_custom.cfg /yolo/recursos/yolov4\_custom\_final.weights

# Script para Inferência - Tensorflow API Object Detection

Depois de carregado o modelo na memória foi adicionado ao *script* a função “*time.time()*”. A alteração destacada em amarelo calcula o tempo de inferência. No Exemplo do terminal do Google Colaboratory:

#CARREGAMENTO DO CAMINHO DA IMAGEM

import os

IMAGE\_PATHS = ""

# Usa uma imagem de teste

IMAGE\_PATHS = '/content/drive/MyDrive/tensorflow/imagem/m11.JPG'

IMAGE\_PATHS

#CARREGAMENTO COMPLETO DO MODELO

%cd /content

import time

import tensorflow as tf # Added as colab instance often crash

from object\_detection.utils import label\_map\_util

from object\_detection.utils import visualization\_utils as viz\_utils

# Caminho do label

PATH\_TO\_LABELS = '/content/drive/MyDrive/tensorflow/workspace/training\_demo/annotations/label\_map.pbtxt'

# Caminho do modelo salvo

PATH\_TO\_SAVED\_MODEL = '/content/drive/MyDrive/tensorflow/workspace/training\_demo/exported-models/1\_faster\_rcnn\_*Inception*\_resnet\_v2\_640/saved\_model'

print('Loading model...', end='')

start\_time = time.time()

# Carrega o modelo e constrói a função de detecção

detect\_fn = tf.saved\_model.load(PATH\_TO\_SAVED\_MODEL)

end\_time = time.time()

elapsed\_time = end\_time - start\_time

print('Done! Took {} seconds'.format(elapsed\_time))

# Conjunto de índices das categorias

category\_index = label\_map\_util.create\_category\_index\_from\_labelmap(PATH\_TO\_LABELS,use\_display\_name=True)

#INFERÊNCIA DE UMA IMAGEM

import numpy as np

from PIL import Image

import matplotlib.pyplot as plt

import warnings

warnings.filterwarnings('ignore')

%matplotlib inline

start\_time = time.time()  #Inicia o tempo de processamento

#Inseri uma imagem para um numpyarray

image\_np = np.array(Image.open(IMAGE\_PATHS))

    # converte em um tensor

input\_tensor = tf.convert\_to\_tensor(image\_np)

    #O modelo espera um batch de imagens

input\_tensor = input\_tensor[tf.newaxis, ...]

    # Input\_tensor = np.expand\_dims(image\_np, 0)

detections = detect\_fn(input\_tensor)

    # Todas as saídas são tensores de lotes.

    # Converta em arraysnumpy e pegue o índice [0] para remover a dimensão do lote.

    # Estamos interessados ​​apenas nos primeiros num\_detections.

num\_detections = int(detections.pop('num\_detections'))

detections = {key: value[0, :num\_detections].numpy()

                   for key, value in detections.items()}

detections['num\_detections'] = num\_detections

# Detecção\_classes deve ser ints.

detections['detection\_classes'] = detections['detection\_classes'].astype(np.int64)

image\_np\_with\_detections = image\_np.copy()

viz\_utils.visualize\_boxes\_and\_labels\_on\_image\_array(

          image\_np\_with\_detections,

          detections['detection\_boxes'],

          detections['detection\_classes'],

          detections['detection\_scores'],

          category\_index,

          use\_normalized\_coordinates=True,

          max\_boxes\_to\_draw=20,

          min\_score\_thresh=.50,

          agnostic\_mode=False)

plt.figure(figsize = (12,8))

plt.imshow(image\_np\_with\_detections)

end\_time = time.time()#Fim do tempo de processamento

elapsed\_time = end\_time - start\_timeCalcula o tempo

print('Done! Took {} seconds'.format(elapsed\_time))

print('Done')

plt.show()

# Script para Inferência - Darknet

Exemplo no terminal do Google Colaboratory:

Célula 1:# Comando para inferir uma imagem a partir do modelo treinado.

!./darknet detector test data/obj.data cfg/yolov2\_custom.cfg /yolo/recursos/yolov2\_custom\_best.weights /yolo/imagens/m11.jpg -thresh 0.5

Célula 2:# Função para mostrar a saída da imagem

import cv2

import matplotlib.pyplot as plt

def mostrar(caminho):

  img = cv2.imread(caminho)

  fig = plt.gcf()

  fig.set\_size\_inches(18, 10)

  plt.axis("off")

  plt.imshow(cv2.cvtColor(img, cv2.COLOR\_BGR2RGB))

  plt.show()

Célula 3:# Mostrar a saída da imagem

mostrar('/content/darknet/predictions.jpg')