Custom Real-Time Object Detection in the Browser Using TensorFlow.js

Hugo Zanini · Follow

Published in Towards Data Science · 14 min read · Jan 26, 2021

> *Train a MobileNetV2 using the TensorFlow 2 Object Detection API and Google Colab, convert the model, and run real-time inferences in the browser through TensorFlow.js.*

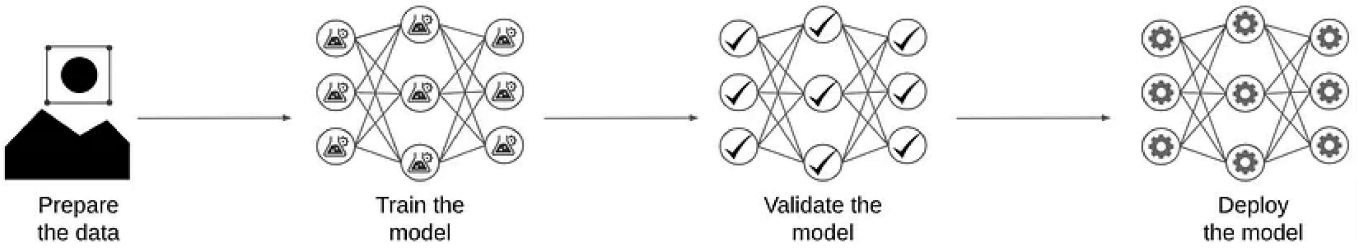
Object detection is the task of detecting and classifying every object of interest in an image. In computer vision, this technique is used in applications such as picture retrieval, security cameras and autonomous vehicles.

One of the most famous families of Deep Convolutional Neural Networks (DNN) for object detection is the YOLO (You Only Look Once), achieving near state-of-the-art results with a single end-to-end model (Redmon, Joseph, et al. — 2016). However, to run it in real-time, it's necessary to have a good graphics card, like an Nvidia Tesla V100 (Bochkovskiy, A., Wang, C. Y., &

Liao, H. Y. M. — 2020), not accessible for everyone. So getting these models into production in a cost-effective way remains a core challenge.

In this post, we are going to develop an end-to-end solution using *TensorFlow* to train a custom object-detection model in *Python*, put it into production, and run real-time inferences in the browser through *TensorFlow.js*. No powerful computers or complex libraries will be needed.

This post is going to be divided into four steps, as follows:



Prepare the data

The first step to train a good model is to have good quality data. When developing this project, I did not find a good and small object detection dataset, so I decided to create my own.

I looked around and saw a Kangaroo sign that I have in my bedroom — a souvenir that I bought to remember my Aussie days. So I decided to build a Kangaroo detector.

To build my dataset, I downloaded 350 kangaroo images from Google and labeled all of them by hand using the LabelImg application. As we can have

more than one animal per image, the process resulted in 520 labeled kangaroos.

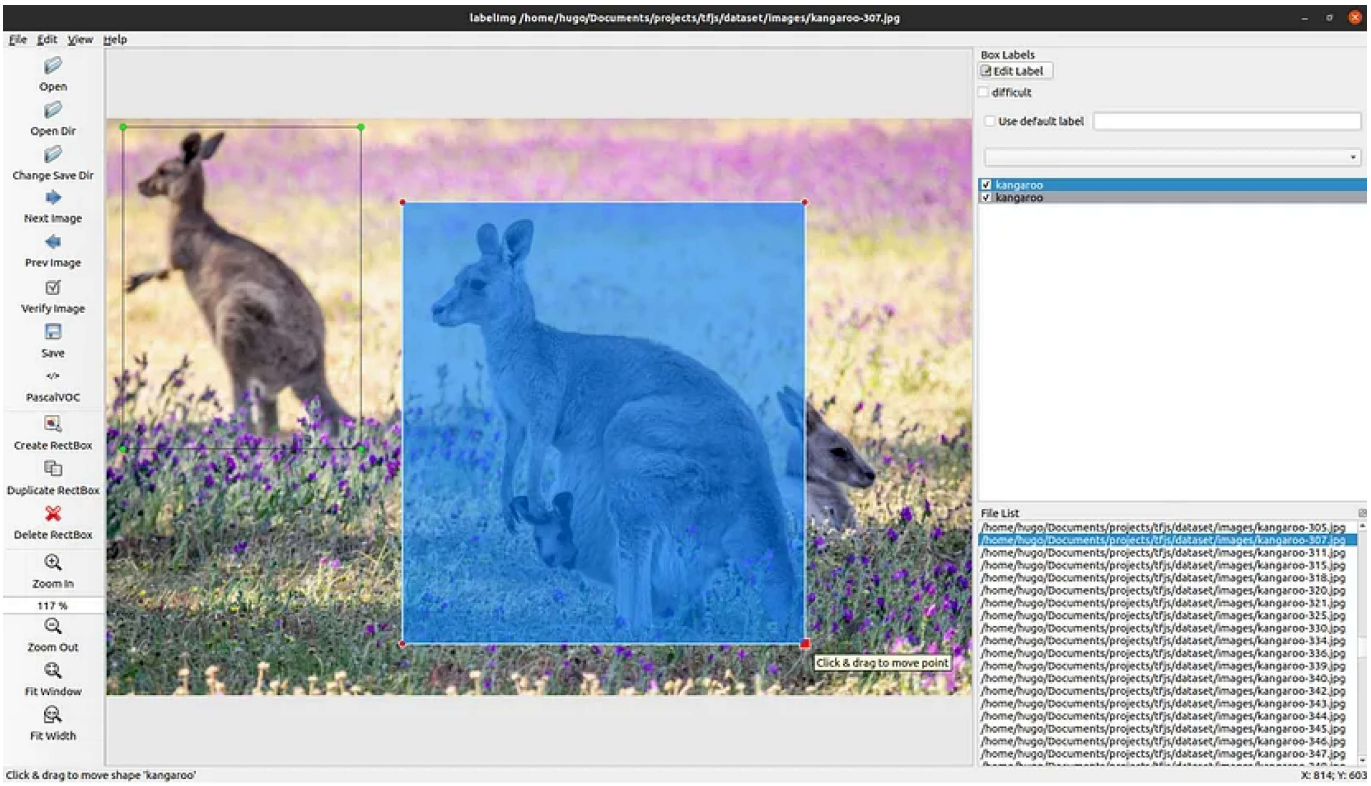


Figure 2: Labelling example (Image by the author)

In that case, I chose just one class, but the software can be used to annotate multiples classes as well. It's going to generate an XML file per image (Pascal VOC format) that contains all annotations and bounding boxes.

```xml
<annotation>
    <folder>images</folder>
    <filename>kangaroo-0.jpg</filename>
    <path>/home/hugo/Documents/projects/tfjs/dataset/images/kangaroo-0.jpg</path>
  <source>
    <database>Unknown</database>
  </source>
  <size>
    <width>3872</width>
    <height>2592</height>
    <depth>3</depth>
  </size>
  <segmented>0</segmented>
  <object>
    <name>kangaroo</name>
    <pose>Unspecified</pose>
    <truncated>0</truncated>
    <difficult>0</difficult>
    <bndbox>
      <xmin>60</xmin>
      <ymin>367</ymin>
      <xmax>2872</xmax>
      <ymax>2399</ymax>
    </bndbox>
  </object>
</annotation>
```

kangaroo-0.xml hosted with ❤ by GitHub view raw

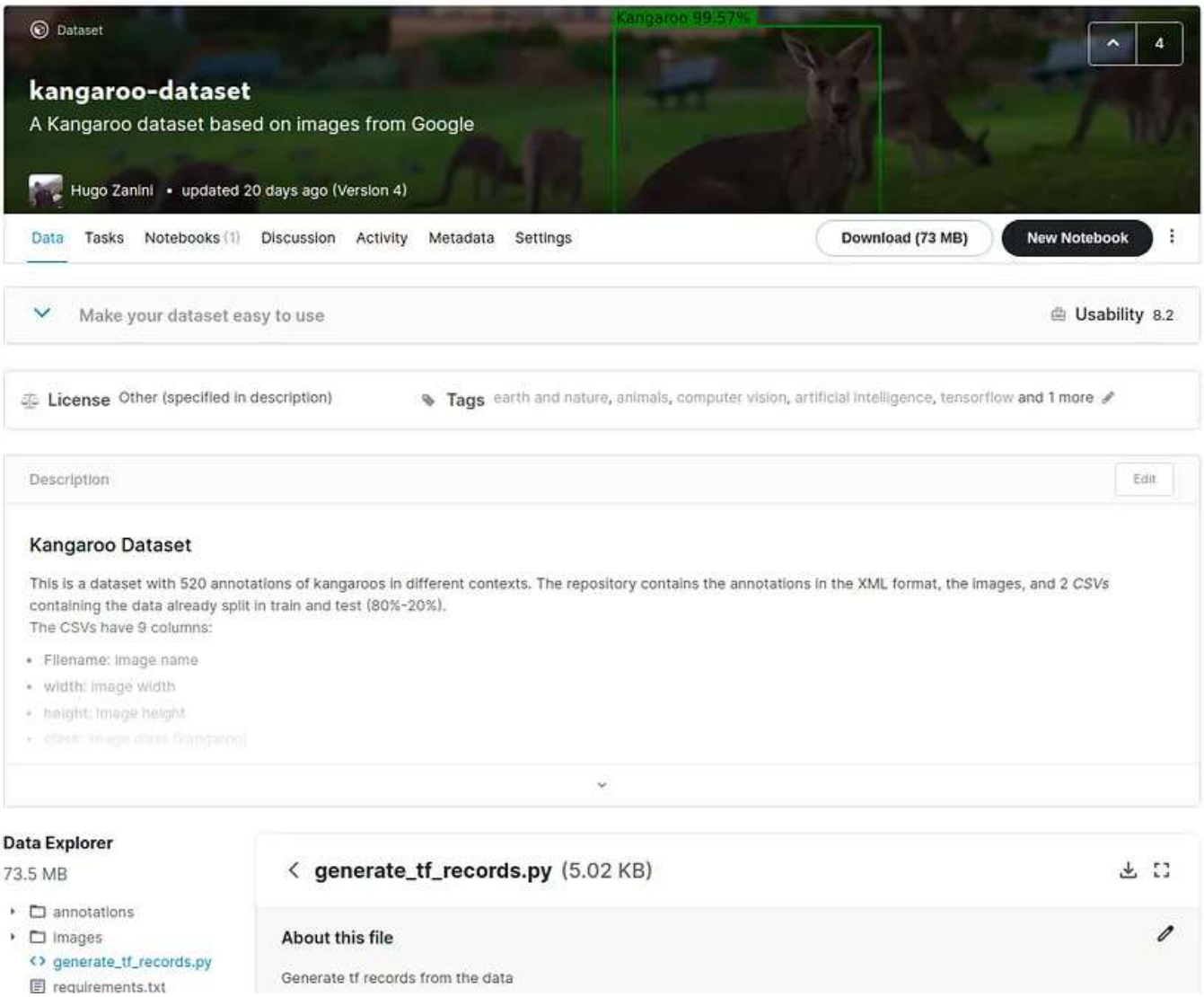
XML Annotation example

To facilitate the manipulation, I've generated two CSV files containing the data already split in train and test (80%-20%). These files have 9 columns:

- **filename:** Image name
- **width:** Image width
- **height:** Image height
- **class:** Image class (kangaroo)

- **xmin**: Minimum bounding box *x* coordinate value
- **ymin**: Minimum bounding box *y* coordinate value
- **xmax**: Maximum value of the *x* coordinate of the bounding box
- **ymax**: Maximum value of the *y* coordinate of the bounding box
- **source**: Image source

Using LabelImg makes it easy to create your own dataset, but feel free to use my kangaroo dataset, I've uploaded it on Kaggle:



Kangaroo Dataset (Image by the author)

Training the model

With a good dataset, it's time to think about the model.TensorFlow 2 provides an Object Detection API that makes it easy to construct, train, and deploy object detection models (Allibhai, E. — 2018). In this project, we're going to use this API and train the model using a Google Colaboratory Notebook. The remainder of this section explains how to set up the environment, the model selection, and training. **If you want to jump straight to the Colab Notebook, click here.**

Setting up the environment

Create a new Google Colab notebook and select a GPU as hardware accelerator:

```
Runtime > Change runtime type > Hardware accelerator: GPU
```

Clone, install, and test the *TensorFlow Object Detection API*:

Clone and install the Tensorflow Object Detection API

In order to use the TensorFlow Object Detection API, we need to clone it's GitHub Repo.

Dependencies

Most of the dependencies required come preloaded in Google Colab. No extra installation is needed.

Protocol Buffers

The TensorFlow Object Detection API relies on what are called `protocol buffers` (also known as `protobufs`). Protobufs are a language neutral way to describe information. That means you can write a protobuf once and then compile it to be used with other languages, like Python, Java or C [5].

The `protoc` command used below is compiling all the protocol buffers in the

install.ipynb hosted with ❤️ by GitHub view raw

Getting and processing the data

As mentioned before, the model is going to be trained using the Kangaroo dataset on Kaggle. If you want to use it as well, it's necessary to create a user, go into the *Account* section, and get an *API Token:*

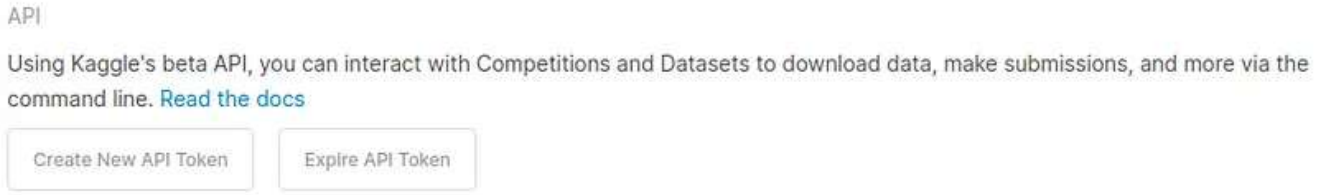


Figure 4: Getting an API token

Then, you're ready to download the data:

```
In [ ]:
!pip install -q kaggle
!pip install -q kaggle-cli
```

```
    |████████████████████████████████| 81kB 6.1MB/s
    |████████████████████████████████| 5.3MB 8.9MB/s
    |████████████████████████████████| 133kB 47.8MB/s
    |████████████████████████████████| 51kB 8.3MB/s
Building wheel for kaggle-cli (setup.py) ... done
Building wheel for PrettyTable (setup.py) ... done
Building wheel for pyperclip (setup.py) ... done
```

```
In [ ]:
import os
os.environ['KAGGLE_USERNAME'] = 'your-username'
os.environ['KAGGLE_KEY'] = 'your-key'
```

```
In [ ]:
%%bash
mkdir /content/dataset
cd /content/dataset
kaggle datasets download -d hugozanini1/kangaroodataset --unzip
```

```
Downloading kangaroodataset.zip to /content/dataset

100%|##########| 72.6M/72.6M [00:01<00:00, 47.1MB/s]
```

getting-data.ipynb hosted with ❤ by GitHub view raw

Now, it's necessary to create a *labelmap* file to define the classes that are going to be used. Kangaroo is the only one, so right-click in the *File* section on *Google Colab* and create a *New file* named `labelmap.pbtxt` as follows:

```
item {
  name: "kangaroo"
  id: 1
}
```

labelmap.pbtxt hosted with ❤ by GitHub view raw

The last step is to convert the data into a sequence of binary records so that they can be fed into *Tensorflow*'s object detection API. To do so, transform the

data into the *TFRecord* format using the `generate_tf_records.py` script available in the Kangaroo Dataset:

```
In [ ]: %cd /content/

/content

In [ ]: !python dataset/generate_tf_records.py -l /content/labelmap.pbtxt -o datas
        !python dataset/generate_tf_records.py -l /content/labelmap.pbtxt -o datas

2020-12-22 18:53:28.918788: I tensorflow/stream_executor/platform/default/dso
_loader.cc:49] Successfully opened dynamic library libcudart.so.10.1
INFO:Successfully created the TFRecords: dataset/train.record
2020-12-22 18:53:31.432478: I tensorflow/stream_executor/platform/default/dso
_loader.cc:49] Successfully opened dynamic library libcudart.so.10.1
INFO:Successfully created the TFRecords: dataset/test.record
```

generate_tf_records.ipynb hosted with ❤️ by GitHub view raw

Choosing the model

We're ready to choose the model that's going to be the Kangaroo Detector. *TensorFlow 2* provides 40 pre-trained detection models on the COCO 2017 Dataset. This collection is the *TensorFlow 2 Detection Model Zoo* and can be accessed here.

Every model has a Speed, Mean Average Precision(mAP) and Output. Generally, a higher mAP implies a lower speed, but as this project is based

on a one-class object detection problem, the faster model (*SSD MobileNet v2 320x320*) should be enough.

Besides the Model Zoo, *TensorFlow* provides a Models Configs Repository as well. There, it's possible to get the configuration file that has to be modified before the training. Let's download the files:

```
In [ ]:
%cd /content
!wget http://download.tensorflow.org/models/object_detection/classifica
!tar -xvf mobilenet_v2.tar.gz
!rm mobilenet_v2.tar.gz
```

```
/content
mobilenet_v2.tar.gz 100%[===================>]   8.01M  30.8MB/s    in 0.3
s

2020-12-22 19:20:36 (30.8 MB/s) - 'mobilenet_v2.tar.gz' saved [8404070/840
4070]

mobilenet_v2/
mobilenet_v2/mobilenet_v2.ckpt-1.index
mobilenet_v2/checkpoint
mobilenet_v2/mobilenet_v2.ckpt-1.data-00001-of-00002
mobilenet_v2/mobilenet_v2.ckpt-1.data-00000-of-00002
```

```
In [ ]:
!wget https://raw.githubusercontent.com/tensorflow/models/master/resear
!mv ssd_mobilenet_v2_320x320_coco17_tpu-8.config mobilenet_v2.config
```

```
ssd_mobilenet_v2_32 100%[===================>]   4.38K  --.-KB/s    in 0s

2020-12-22 19:20:42 (77.4 MB/s) - 'ssd_mobilenet_v2_320x320_coco17_tpu-8.c
onfig' saved [4484/4484]
```

getting-weights-and-config.ipynb hosted with ❤ by GitHub view raw

Configuring training

As mentioned before, the downloaded weights were pre-trained on the COCO 2017 Dataset, but the focus here is to train the model to recognize one class so these weights are going to be used only to initialize the network —

this technique is known as transfer learning, and it's commonly used to speed up the learning process.

From now, what has to be done is to set up the *mobilenet_v2.config* file, and start the training. I highly recommend reading the *MobileNetV2 paper* (Sandler, Mark, et al. — 2018) to get the gist of the architecture.

Choosing the best hyperparameters is a task that requires some experimentation. As the resources are limited in the Google Colab, I am going to use the same batch size as the paper, set a number of steps to get a reasonably low loss, and leave all the other values as default. If you want to try something more sophisticated to find the hyperparameters, I recommend Keras Tuner — an easy-to-use framework that applies Bayesian Optimization, Hyperband, and Random Search algorithms.

Defining training parameters

In [ ]:

```
num_classes = 1
batch_size = 96
num_steps = 7500
num_eval_steps = 1000

train_record_path = '/content/dataset/train.record'
test_record_path = '/content/dataset/test.record'
model_dir = '/content/training/'
labelmap_path = '/content/labelmap.pbtxt'

pipeline_config_path = 'mobilenet_v2.config'
fine_tune_checkpoint = '/content/mobilenet_v2/mobilenet_v2.ckpt-1'
```

Editing config file

In [ ]:

```
import re

with open(pipeline_config_path) as f:
    config = f.read()

with open(pipeline_config_path, 'w') as f:
```

setting-paramerts.ipynb hosted with ❤️ by GitHub view raw

With the parameters set, start the training:

```
tep time 2.494s loss=0.310
INFO:tensorflow:Step 6700 per-step time 2.443s loss=0.268
I1220 16:56:13.485662 140535951021952 model_lib_v2.py:651] Step 6700 per-s
tep time 2.443s loss=0.308
INFO:tensorflow:Step 6800 per-step time 2.363s loss=0.293
I1220 17:00:22.396368 140535951021952 model_lib_v2.py:651] Step 6800 per-s
tep time 2.363s loss=0.294
INFO:tensorflow:Step 6900 per-step time 2.287s loss=0.304
I1220 17:04:32.806847 140535951021952 model_lib_v2.py:651] Step 6900 per-s
tep time 2.287s loss=0.293
INFO:tensorflow:Step 7000 per-step time 2.484s loss=0.284
I1220 17:08:44.175837 140535951021952 model_lib_v2.py:651] Step 7000 per-s
tep time 2.484s loss=0.304
INFO:tensorflow:Step 7100 per-step time 2.257s loss=0.310
I1220 17:12:55.832094 140535951021952 model_lib_v2.py:651] Step 7100 per-s
tep time 2.257s loss=0.293
INFO:tensorflow:Step 7200 per-step time 2.470s loss=0.328
I1220 17:17:06.899618 140535951021952 model_lib_v2.py:651] Step 7200 per-s
tep time 2.470s loss=0.284
INFO:tensorflow:Step 7300 per-step time 2.376s loss=0.281
I1220 17:21:16.386411 140535951021952 model_lib_v2.py:651] Step 7300 per-s
tep time 2.376s loss=0.281
INFO:tensorflow:Step 7400 per-step time 2.312s loss=0.275
I1220 17:25:26.509467 140535951021952 model_lib_v2.py:651] Step 7400 per-s
tep time 2.312s loss=0.275
INFO:tensorflow:Step 7500 per-step time 2.525s loss=0.308
I1220 17:29:36.155840 140535951021952 model_lib_v2.py:651] Step 7500 per-s
tep time 2.525s loss=0.268
```

training.ipynb hosted with ❤️ by GitHub view raw

To identify how well the training is going, we use the loss value. Loss is a number indicating how bad the model's prediction was on the training samples. If the model's prediction is perfect, the loss is zero; otherwise, the loss is greater. The goal of training a model is to find a set of weights and biases that have low loss, on average, across all examples (Descending into ML: Training and Loss | Machine Learning Crash Course).

From the logs, it's possible to see a downward trend in the values so we say that "*The model is converging*". In the next section, we're going to plot these values for all training steps and the trend will be even clearer.

The model took around 4h to train, but by setting different parameters, you can make the process faster or slower. Everything depends on the number of classes you are using and your Precision/Recall target. A highly accurate network that recognizes multiple classes will take more steps and require more detailed parameters tuning.

Validate the model

Now let's evaluate the trained model using the test data:

Here we're going to run the code through a loop that waits for checkpoints to evaluate. Once the evaluation finishes, you're going to see the message:

```
INFO:tensorflow:Waiting for new checkpoint at
/content/training/
```

Then you can stop the cell

In [ ]:

```
!python /content/models/research/object_detection/model_main_tf2.py \
    --pipeline_config_path={pipeline_config_path} \
    --model_dir={model_dir} \
    --checkpoint_dir={model_dir}
```

```
INFO:tensorflow:Performing evaluation on 89 images.
I1220 17:34:19.304409 139900847355776 coco_evaluation.py:293] Performing e
valuation on 89 images.
creating index...
index created!
INFO:tensorflow:Loading and preparing annotation results...
I1220 17:34:19.304893 139900847355776 coco_tools.py:116] Loading and prepa
ring annotation results...
INFO:tensorflow:DONE (t=0.00s)
I1220 17:34:19.308917 139900847355776 coco_tools.py:138] DONE (t=0.00s)
creating index...
index created!
Running per image evaluation
```

validation.ipynb hosted with ❤️ by GitHub view raw

The evaluation was done in 89 images and provides three metrics based on the COCO detection evaluation metrics: Precision, Recall and Loss.

The Recall measures how good the model is at hitting the positive class, That is, **from the positive samples, how many did the algorithm get right?**

$$R = \frac{TP}{TP + FN}$$

Recall

Precision defines how much you can rely on the positive class prediction: **From the samples that the model said were positive, how many actually are?**

$$P = \frac{TP}{TP + FP}$$

Precision

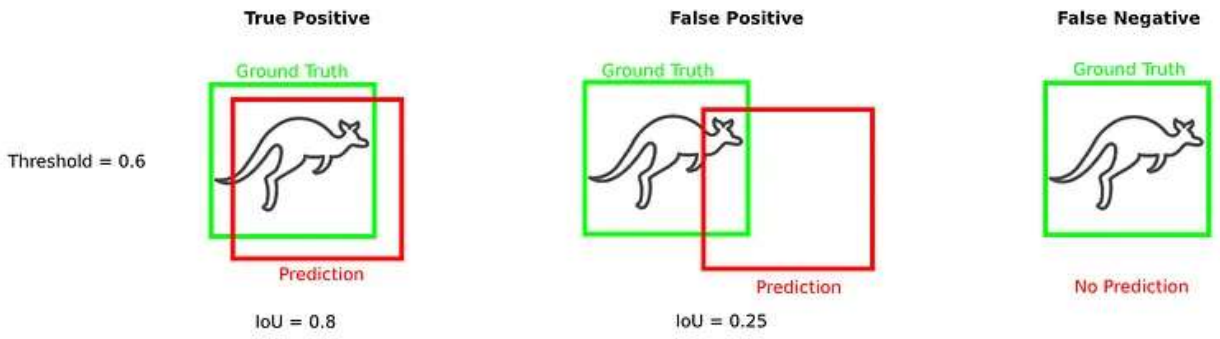
Setting a practical example: Imagine we have an image containing 10 kangaroos, our model returned 5 detections, being 3 real kangaroos (TP = 3, FN =7) and 2 wrong detections (FP = 2). In that case, we have a 30% recall (the model detected 3 out of 10 kangaroos in the image) and a 60% precision (from the 5 detections, 3 were correct).

The precision and recall were divided by Intersection over Union (IoU) thresholds. The IoU is defined as the area of the intersection divided by the area of the union of a predicted bounding box (B_p) to a ground-truth box (B_t) (Zeng, N. — 2018):

$$IoU = \frac{area(B_p \cap B_t)}{area(B_p \cup B_t)}$$

Intersection over Union

For simplicity, it's possible to consider that the IoU thresholds are used to determine whether a detection is a true positive(TP), a false positive(FP) or a false negative (FN). See an example below:



IoU threshold examples (Image by the author)

With these concepts in mind, we can analyze some of the metrics we got from the evaluation. From the TensorFlow 2 Detection Model Zoo, the *SSD MobileNet v2 320x320* has an mAP of 0.202. Our model presented the following average precisions (AP) for different IoUs:

```
AP@[IoU=0.50:0.95 | area=all | maxDets=100] = 0.222
AP@[IoU=0.50      | area=all | maxDets=100] = 0.405
AP@[IoU=0.75      | area=all | maxDets=100] = 0.221
```

That's pretty good! And we can compare the obtained APs with the *SSD MobileNet v2 320x320* mAP as from the COCO Dataset documentation:

> *We make no distinction between AP and mAP (and likewise AR and mAR) and assume the difference is clear from context.*

The Average Recall(AR) was split by the max number of detection per image (1, 10, 100). When we have just one kangaroo per image, the recall is around 30% while when we have up to 100 kangaroos it is around 51%. These values are not that good but are reasonable for the kind of problem we're trying to solve.

```
(AR)@[ IoU=0.50:0.95 | area=all | maxDets=  1] = 0.293
(AR)@[ IoU=0.50:0.95 | area=all | maxDets= 10] = 0.414
(AR)@[ IoU=0.50:0.95 | area=all | maxDets=100] = 0.514
```

The Loss analysis is very straightforward, we've got 4 values:

```
INFO:tensorflow: + Loss/localization_loss: 0.345804
INFO:tensorflow: + Loss/classification_loss: 1.496982
INFO:tensorflow: + Loss/regularization_loss: 0.130125
INFO:tensorflow: + Loss/total_loss: 1.972911
```

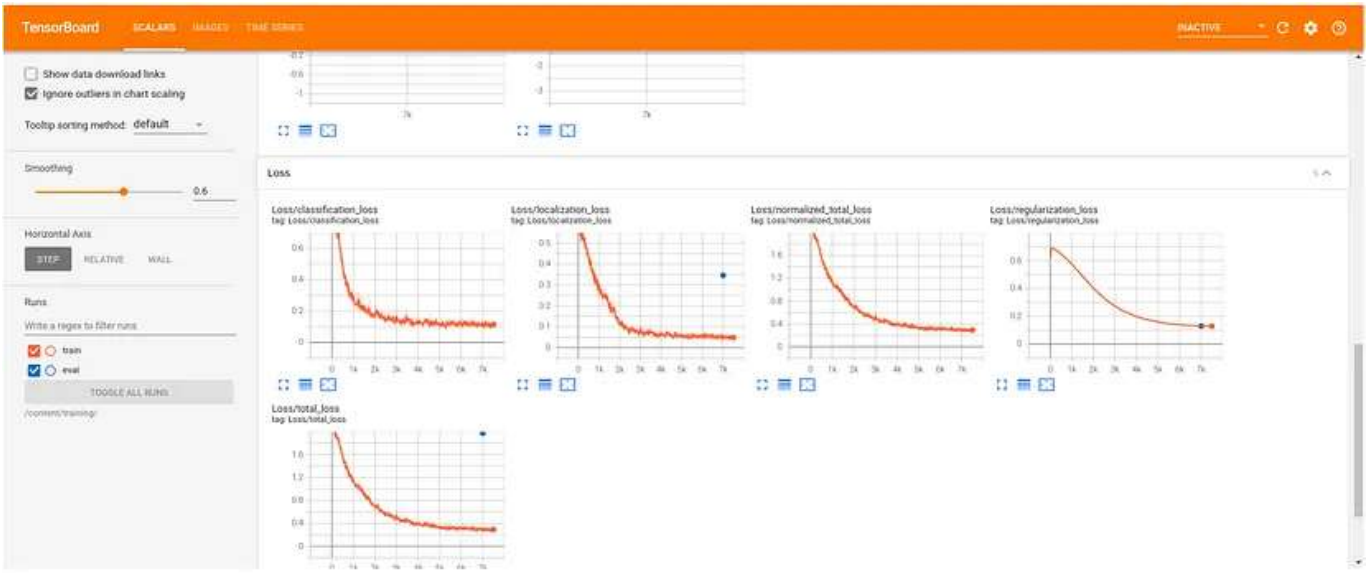
The localization loss computes the difference between the predicted bounding boxes and the labeled ones. The classification loss indicates whether the bounding box class matches with the predicted class. The regularization loss is generated by the network's regularization function and helps to drive the optimization algorithm in the right direction. The last term is the total loss and is the sum of three previous ones.

Tensorflow provides a tool to visualize all these metrics in an easy way. It's called TensorBoard and can be initialized by the following command:

```
%load_ext tensorboard
```

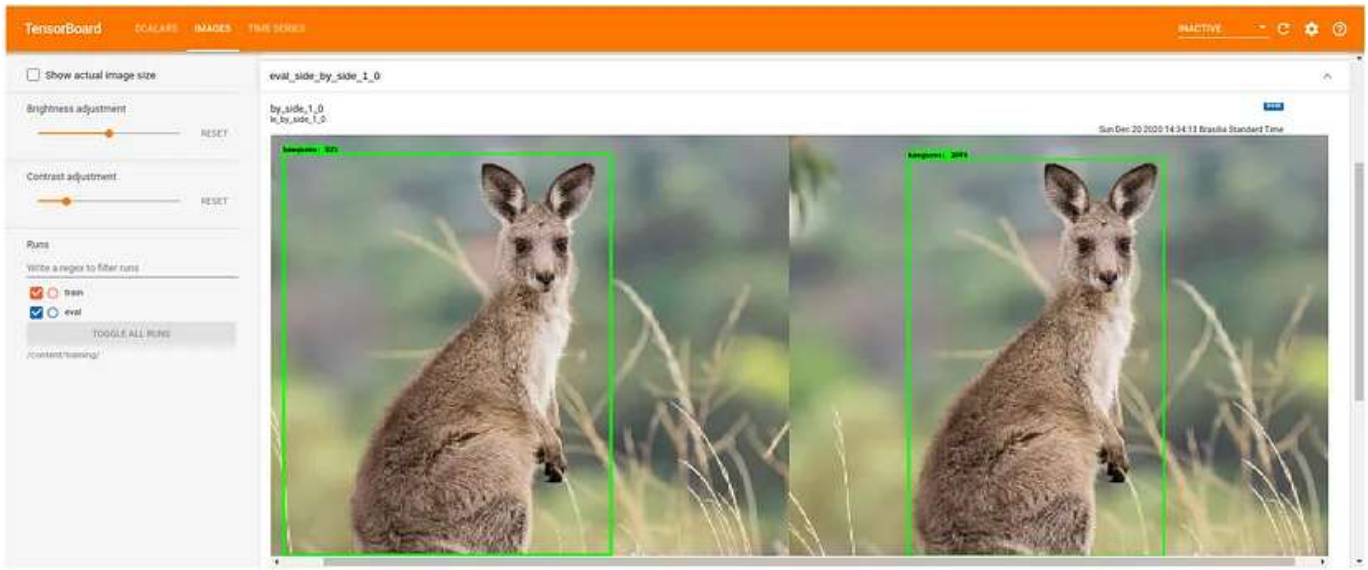
```
%tensorboard --logdir '/content/training/'
```

This is going to be shown, and you can explore all training and evaluation metrics.



Tensorboard — Loss (Image by the author)

In the tab `IMAGES` , it's possible to find some comparisons between the predictions and the ground truth side by side. A very interesting resource to explore during the validation process as well.



Tensorboard — Testing images (Image by the author)

Exporting the model

Now that the training is validated, it's time to export the model. We're going to convert the training checkpoints to a *protobuf (pb)* file. This file is going to have the graph definition and the weights of the model.

Export the Inference Graph

The below code cell adds a line to the tf_utils.py file. This is a temporary fix to a exporting issue occuring when using the API with Tensorflow 2. This code will be removed as soon as the TF Team puts out a fix.

All credit goes to the Github users Jacobsolawetz and Tanner Gilbert, who provided this temporary fix.

In [ ]:

```
with open('/usr/local/lib/python3.6/dist-packages/tensorflow/python/ker
    tf_utils = f.read()

with open('/usr/local/lib/python3.6/dist-packages/tensorflow/python/ker
  # Set labelmap path
  throw_statement = "raise TypeError('Expected Operation, Variable, or
  tf_utils = tf_utils.replace(throw_statement, "if not isinstance(x, st
  f.write(tf_utils)
```

In [ ]:

```
output_directory = 'inference_graph'

!python /content/models/research/object_detection/exporter_main_v2.py \
    --trained_checkpoint_dir {model_dir} \
    --output_directory {output_directory} \
```

exporting-pb.ipynb hosted with ❤️ by GitHub view raw

As we're going to deploy the model using *TensorFlow.js* and *Google Colab* has a maximum lifetime limit of 12 hours, let's download the trained weights and save them locally.

Downloading weights

In [ ]:
```
!zip -r /content/saved_model.zip /content/inference_graph/saved_model/
```

```
  adding: content/inference_graph/saved_model/ (stored 0%)
  adding: content/inference_graph/saved_model/saved_model.pb (deflated 93%)
  adding: content/inference_graph/saved_model/assets/ (stored 0%)
  adding: content/inference_graph/saved_model/variables/ (stored 0%)
  adding: content/inference_graph/saved_model/variables/variables.data-00000-
of-00001 (deflated 7%)
  adding: content/inference_graph/saved_model/variables/variables.index (defl
ated 76%)
```

In [ ]:
```
from google.colab import files
files.download("/content/saved_model.zip")
```

downloading-pb.ipynb hosted with ❤️ by GitHub view raw

If you want to check if the model was saved properly, load, and test it. I've created some functions to make this process easier so feel free to clone the `inferenceutils.py` file from my GitHub to test some images.

Testing the trained model

Based on Object Detection API Demo and Inference from saved model tf2 colab.

In [ ]:
```
!wget https://raw.githubusercontent.com/hugozanini/object-detection/mas
from inferenceutils import *
```

```
--2020-12-24 15:30:44--  https://raw.githubusercontent.com/hugozanini/obje
ct-detection/master/inferenceutils.py
Resolving raw.githubusercontent.com (raw.githubusercontent.com)... 151.10
1.0.133, 151.101.64.133, 151.101.128.133, ...
Connecting to raw.githubusercontent.com (raw.githubusercontent.com)|151.10
1.0.133|:443... connected.
HTTP request sent, awaiting response... 200 OK
Length: 2219 (2.2K) [text/plain]
Saving to: ‘inferenceutils.py’

inferenceutils.py   100%[===================>]   2.17K  --.-KB/s    in 0s

2020-12-24 15:30:44 (46.5 MB/s) - ‘inferenceutils.py’ saved [2219/2219]
```

Loading the model

In [ ]:
```
output_directory = 'inference_graph/'
```

testing.ipynb hosted with ❤ by GitHub view raw

Everything is working well, so we’re ready to put the model in production.

Deploying the model

The model is going to be deployed in a way that anyone can open a PC or mobile camera and perform inferences in real-time through a web browser. To do that, we’re going to convert the saved model to the Tensorflow.js layers format, load the model in a javascript application and make everything available on Glitch.

Converting the model

At this point, you should have something similar to this structure saved locally:

```
├── inference-graph
| ├── saved_model
| | ├── assets
| | ├── saved_model.pb
| | ├── variables
| | ├── variables.data-00000-of-00001
| | └── variables.index
```

Open a terminal in the inference-graph folder and create and activate a new virtual environment:

```
virtualenv -p python3 venv
source venv/bin/activate
```

Install the TensorFlow.js converter:

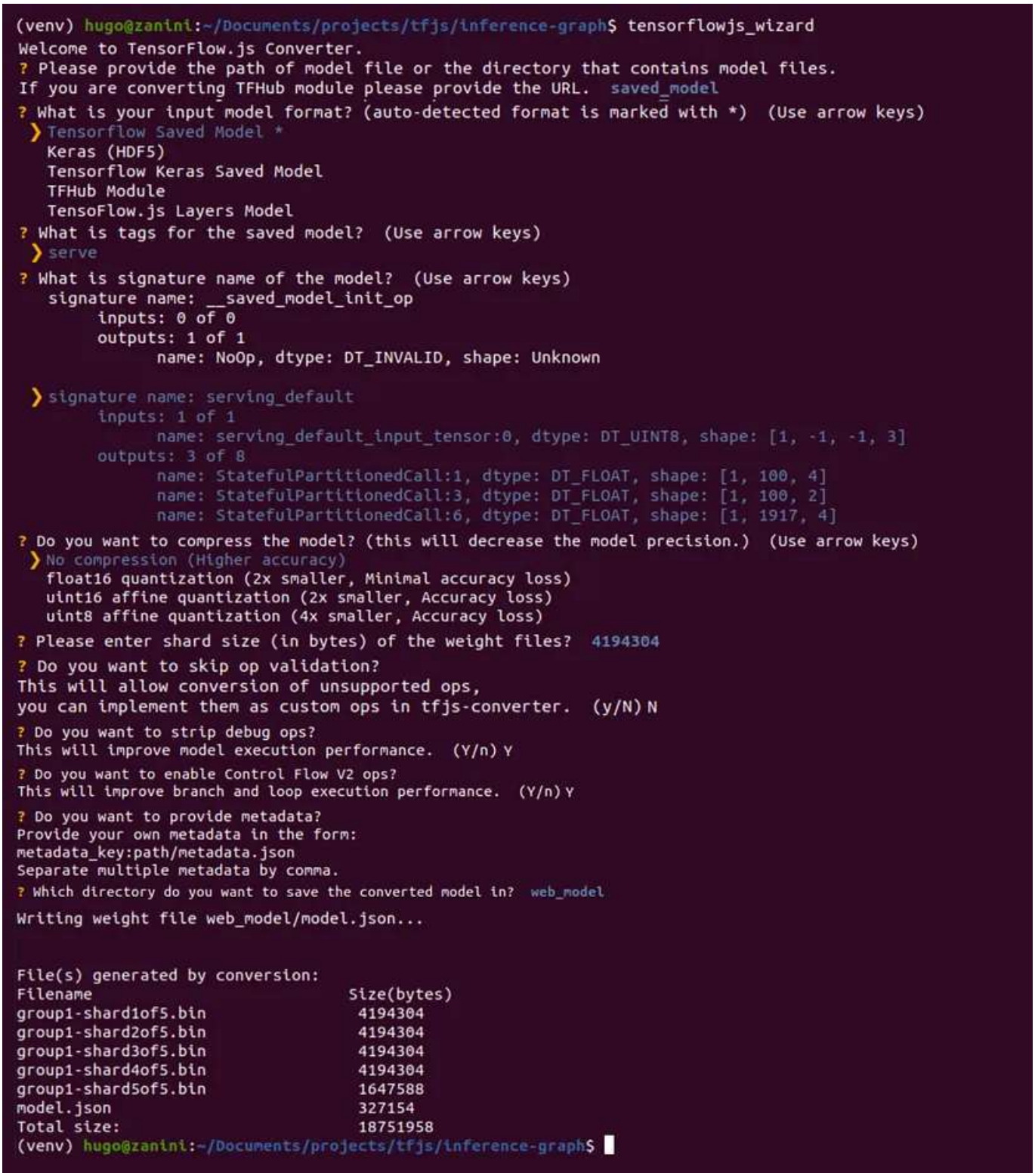
```
pip install tensorflowjs[wizard]
```

Start the conversion wizard:

```
tensorflowjs_wizard
```

Now, the tool will guide you through the conversion, providing explanations for each choice you need to make. Figure 8 shows all the choices that were made to convert the model. Most of them are the standard ones, but options like the shard sizes and compression can be changed according to your needs.

To enable the browser to cache the weights automatically, it's recommended to split them into shard files of around 4MB. To guarantee that the conversion is going to work, don't skip the op validation as well, not all *TensorFlow* operations are supported so some models can be incompatible with *TensorFlow.js* — See this list for which ops are currently supported.



Model conversion using Tensorflow.js Converter (Full resolution image here — Image by the author)

If everything worked well, you're going to have the model converted to the *Tensorflow.js* layers format in the `web_model` directory. The folder contains a `model.json` file and a set of sharded weights files in a binary format. The `model.json` has both the model topology (aka "architecture" or "graph": a description of the layers and how they are connected) and a manifest of the weight files (Lin, Tsung-Yi, et al).

```
└ web_model
  ├── group1-shard1of5.bin
  ├── group1-shard2of5.bin
  ├── group1-shard3of5.bin
  ├── group1-shard4of5.bin
  ├── group1-shard5of5.bin
  └── model.json
```

Configuring the application

The model is ready to be loaded in javascript. I've created an application to perform inferences directly from the browser. Let's clone the repository to figure out how to use the converted model in real-time. This is the project structure:

```
├── models
│   └── kangaroo-detector
│       ├── group1-shard1of5.bin
│       ├── group1-shard2of5.bin
│       ├── group1-shard3of5.bin
│       ├── group1-shard4of5.bin
│       ├── group1-shard5of5.bin
│       └── model.json
├── package.json
├── package-lock.json
├── public
│   └── index.html
├── README.MD
```

```
└── src
    ├── index.js
    └── styles.css
```

For the sake of simplicity, I already provide a converted `kangaroo-detector` model in the `models` folder. However, let's put the `web_model` generated in the previous section in the `models` folder and test it.

The first thing to do is to define how the model is going to be loaded in the function `load_model` (lines 10–15 in the file `src>index.js`). There are two choices.

The first option is to create an *HTTP server* locally that will make the model available in a URL allowing requests and be treated as a REST API. When loading the model, *TensorFlow.js* will do the following requests:

```
GET /model.json
GET /group1-shard1of5.bin
GET /group1-shard2of5.bin
GET /group1-shard3of5.bin
GET /group1-shardo4f5.bin
GET /group1-shardo5f5.bin
```

If you choose this option, define the `load_model` function as follows:

```
1  async function load_model() {
2      // It's possible to load the model locally or from a repo
3      const model = await loadGraphModel("http://127.0.0.1:8080/model.json");
4      //const model = await loadGraphModel("https://raw.githubusercontent.com/hugozanini/TFJS-object
5      return model;
6    }
```

load_model_locally.js hosted with ❤ by GitHub view raw

Then install the http-server:

```
npm install http-server -g
```

Go to `models > web_model` and run the command below to make the model available at `http://127.0.0.1:8080` . This a good choice when you want to keep the model weights in a safe place and control who can request inferences to it.

```
http-server -c1 --cors .
```

The second option is to upload the model files somewhere, in my case, I chose my own Github repo and referenced to the `model.json` URL in the `load_model` function:

```
async function load_model() {
    // It's possible to load the model locally or from a repo
    //const model = await loadGraphModel("http://127.0.0.1:8080/model.json");
    const model = await loadGraphModel("https://raw.githubusercontent.com/hugozanini/TFJS-object-
    return model;
  }
```

load_model.js hosted with ❤ by GitHub view raw

This is a good option because it gives more flexibility to the application and makes it easier to run on some platform as Glitch.

Running locally

To run the app locally, install the required packages:

```
npm install
```

And start:

```
npm start
```

The application is going to run at `http://localhost:3000` and you should see something similar to this:

Application running locally (Image by the author)

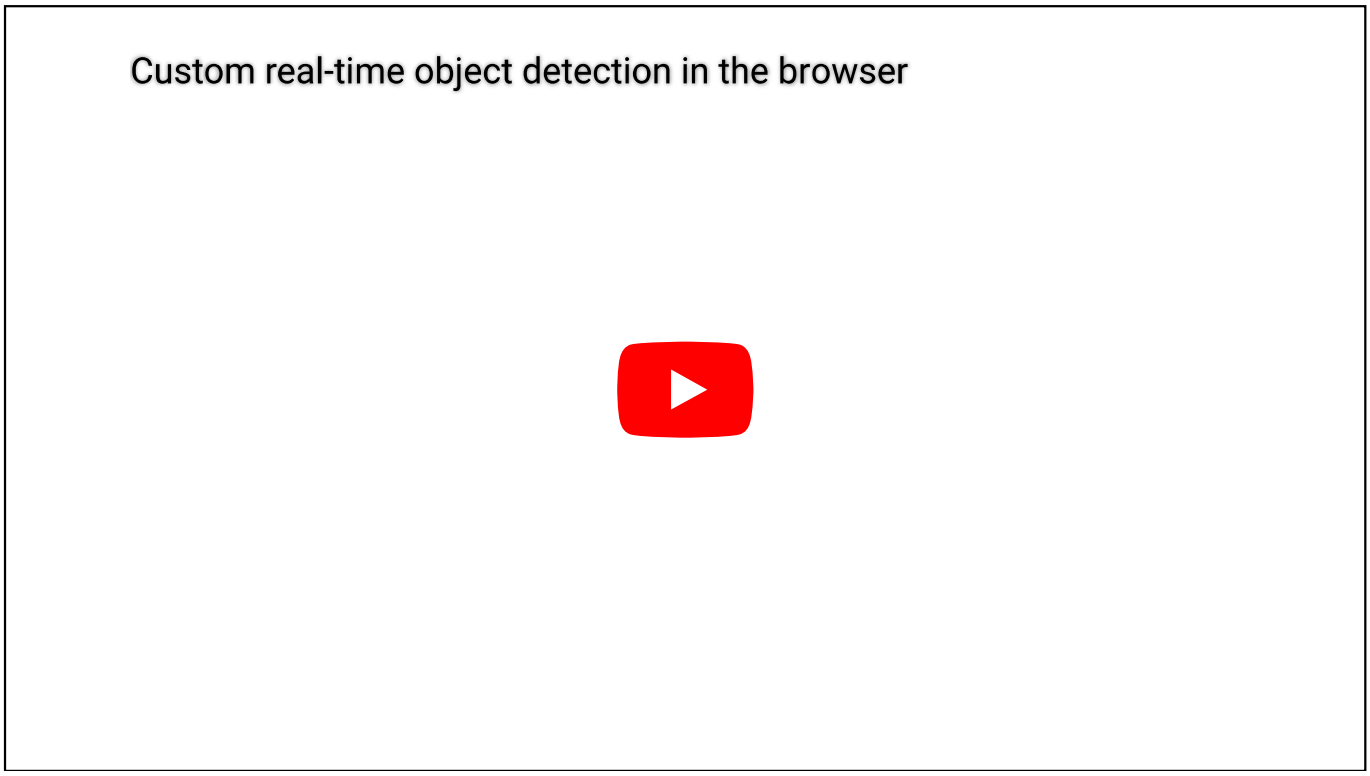
The model takes from 1 to 2 seconds to load and, after that, you can show kangaroos images to the camera and the application is going to draw

bounding boxes around them.

Publishing in Glitch

Glitch is a simple tool for creating web apps where we can upload the code and make the application available for everyone on the web. Uploading the model files in a GitHub repo and referencing to them in the `load_model` function, we can simply log into *Glitch*, click on `New project > Import from Github` and select the app repository.

Wait some minutes to install the packages and your app will be available in a public URL. Click on `Show > In a new window` and a tab will be open. Copy this URL and past it in any web browser (PC or Mobile) and your object detection will be ready to run. See some examples in the video below:



Running the model on different devices (Video by the author)

First, I did a test showing a kangaroo sign to verify the robustness of the application. It showed that the model is focusing specifically on the kangaroo features and did not specialize in irrelevant characteristics that were present in many images, such as pale colors or shrubs.

Then, I opened the app on my mobile and showed some images from the test set. The model runs smoothly and identifies most of the kangaroos. If you want to test my live app, it is available here (glitch takes some minutes to wake up).

Besides the accuracy, an interesting part of these experiments is the inference time — everything runs in real-time. Good object detection models running in the browser and using few computational resources is a must in many applications, mostly in the industry. Putting the Machine Learning model on the client-side means cost reduction and safer applications, once there is no need to send the information to any server to perform the inferences.

Next steps

Custom real-time object detection in the browser can solve a lot of real-world problems and I hope this article will serve as a basis for new projects involving Computer Vision, Python, TensorFlow and Javascript.

As the next steps, I'd like to make more detailed training experiments. Due to the lack of resources, I could not try many different parameters and I'm sure that there is a lot of room for improvements in the model.

I'm more focused on the models' training, but I'd like to see a better user interface for the app. If someone is interested in contributing to the project,

feel free to create a pull request in the project repo. It will be nice to make a more user-friendly application.

If you have any questions or suggestions you can reach me out on Linkedin. Thanks for reading!

References

[0] Redmon, J., Divvala, S., Girshick, R., & Farhadi, A. (2016). You only look once: Unified, real-time object detection. In *Proceedings of the IEEE conference on computer vision and pattern recognition* (pp. 779–788).

[1] Bochkovskiy, A., Wang, C. Y., & Liao, H. Y. M. (2020). YOLOv4: Optimal Speed and Accuracy of Object Detection. *arXiv preprint arXiv:2004.10934*.

[2] Allibhai, E. (2018, November 21). Building A Deep Learning Model using Keras. Retrieved December 28, 2020, from https://towardsdatascience.com/building-a-deep-learning-model-using-keras-1548ca149d37

[3] Sandler, M., Howard, A., Zhu, M., Zhmoginov, A., & Chen, L. C. (2018). Mobilenetv2: Inverted residuals and linear bottlenecks. In Proceedings of the IEEE conference on computer vision and pattern recognition (pp. 4510–4520).

[4] Descending into ML: Training and Loss | Machine Learning Crash Course. (n.d.). Retrieved December 28, 2020, from https://developers.google.com/machine-learning/crash-course/descending-into-ml/training-and-loss

[5] Zeng, N. (2018, December 16). An Introduction to Evaluation Metrics for Object Detection: NickZeng: 曾广宇. Retrieved December 28, 2020, from https://blog.zenggyu.com/en/post/2018-12-16/an-introduction-to-evaluation-metrics-for-object-detection/

[6] Lin, Tsung-Yi, et al. “Microsoft coco: Common objects in context.” European conference on computer vision. Springer, Cham, 2014. Retrieved December 28, 2020, from https://cocodataset.org/#detection-eval

[7] Importing a Keras model into TensorFlow.js. (2020, March 31). Retrieved December 28, 2020, from https://www.tensorflow.org/js/tutorials/conversion/import_keras

Object Detection | Tensorflowjs | Machine Learning | Computer Vision | TensorFlow

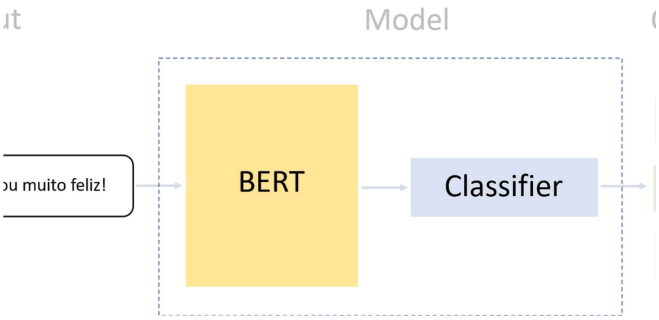
Written by Hugo Zanini

137 Followers · Writer for Towards Data Science

GDE in Machine Learning| Electrical Engineer | Technical Product Manager

Follow

More from Hugo Zanini and Towards Data Science



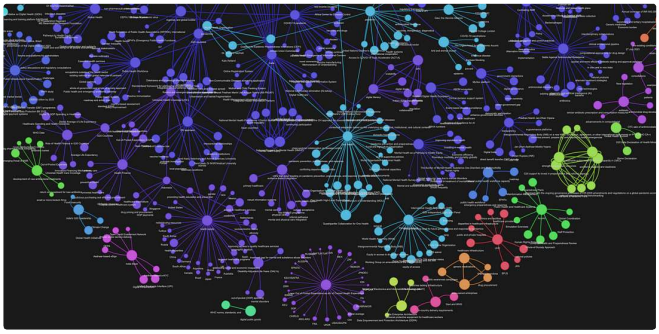
Hugo Zanini in Data Hackers

Análise de sentimentos em português utilizando Pytorch e...

Esse é o primeiro de dois artigos que irão mostrar como criar e produtizar um modelo...

7 min read · Nov 2, 2021

419 4



Rahul Nayak in Towards Data Science

How to Convert Any Text Into a Graph of Concepts

A method to convert any text corpus into a Knowledge Graph using Mistral 7B.

12 min read · Nov 10

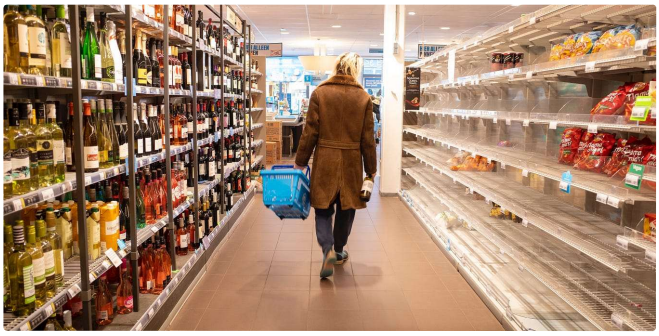
3.6K 39

Anthony Alcaraz in Towards Data Science

Embeddings + Knowledge Graphs: The Ultimate Tools for RAG...

The advent of large language models (LLMs) , trained on vast amounts of text data, has be...

· 10 min read · Nov 14



Hugo Zanini in Towards Data Science

Training a custom YOLOv7 in PyTorch and running it directly in...

Using a YOLOv7 model to recognize empty shelves in real-time, offiline and in the...

6 min read · Mar 29

Predictive Modeling w/ Python

20 stories · 662 saves

Practical Guides to Machine Learning

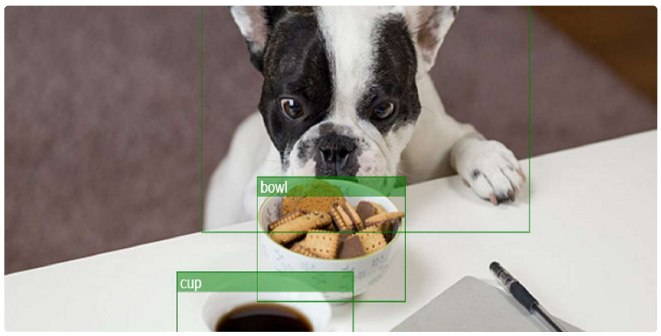
10 stories · 742 saves

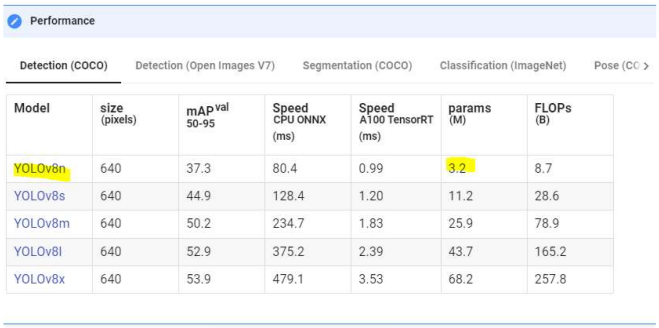
Natural Language Processing

932 stories · 443 saves

The New Chatbots: ChatGPT, Bard, and Beyond

12 stories · 227 saves





NEERAJ VAGEELE

Object detection in the image using TensorFlow in NextJS

Object detection is a technique of identifying objects in images or videos using machine...

3 min read · Jun 19

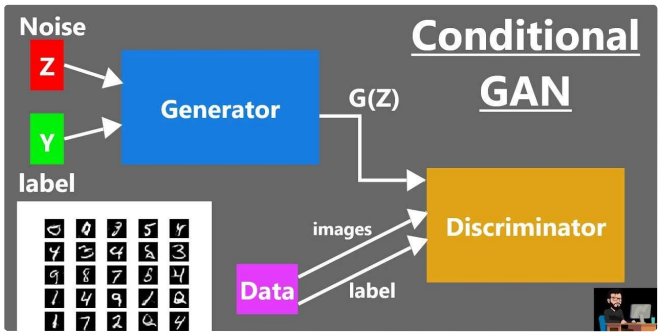
2

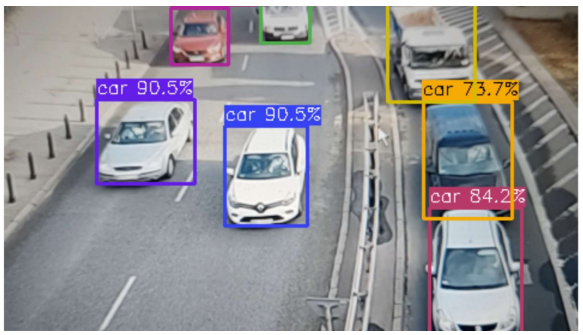
Ekkachai

YOLOv8 — Detection from Webcam — Step by Step [CPU]

This article shows how to use YOLOv8 for object detection with a web camera.

4 min read · Nov 9





Francesco Franco in GoPenAI

Ramesh Pokhrel

Conditional GAN with Tensorflow and Keras

In a previous post, we discussed Generative Adversarial Networks (GANs) in some detail...

9 min read · Nov 16

5

Object Detection and Tracking in Android (Native C++)- Part 1

This will be a series of discussions on detection and tracking in Android. I will cove...

13 min read · Jun 9

36

See more recommendations