Building your own Object Detector from scratch with Tensorflow

Luiz doleron · Follow
Published in MLearning.ai
10 min read · Mar 31

In this story, we talk about how to build a **Deep Learning Object Detector from scratch** using **TensorFlow.** Instead of using a predefined model, we will define each

layer in the network and then we will train our model to detect both the object bound box and its class. Finally, we will evaluate the model using IoU metric. Everything in a surprisingly easy and understandable way.

> *TL;DR: need the code right now? Check this colab notebook or this github repository*

Object Detection Premier

Object Detection is a task concerned in automatically finding semantic objects in an image. Today Object Detectors like YOLO v4/v5 /v7 and v8 achieve state-of-art in terms of accuracy at impressive real time FPS rate.

> *Check one of my previous stories if you want to learn how to use YOLOv5 with Python or C++.*

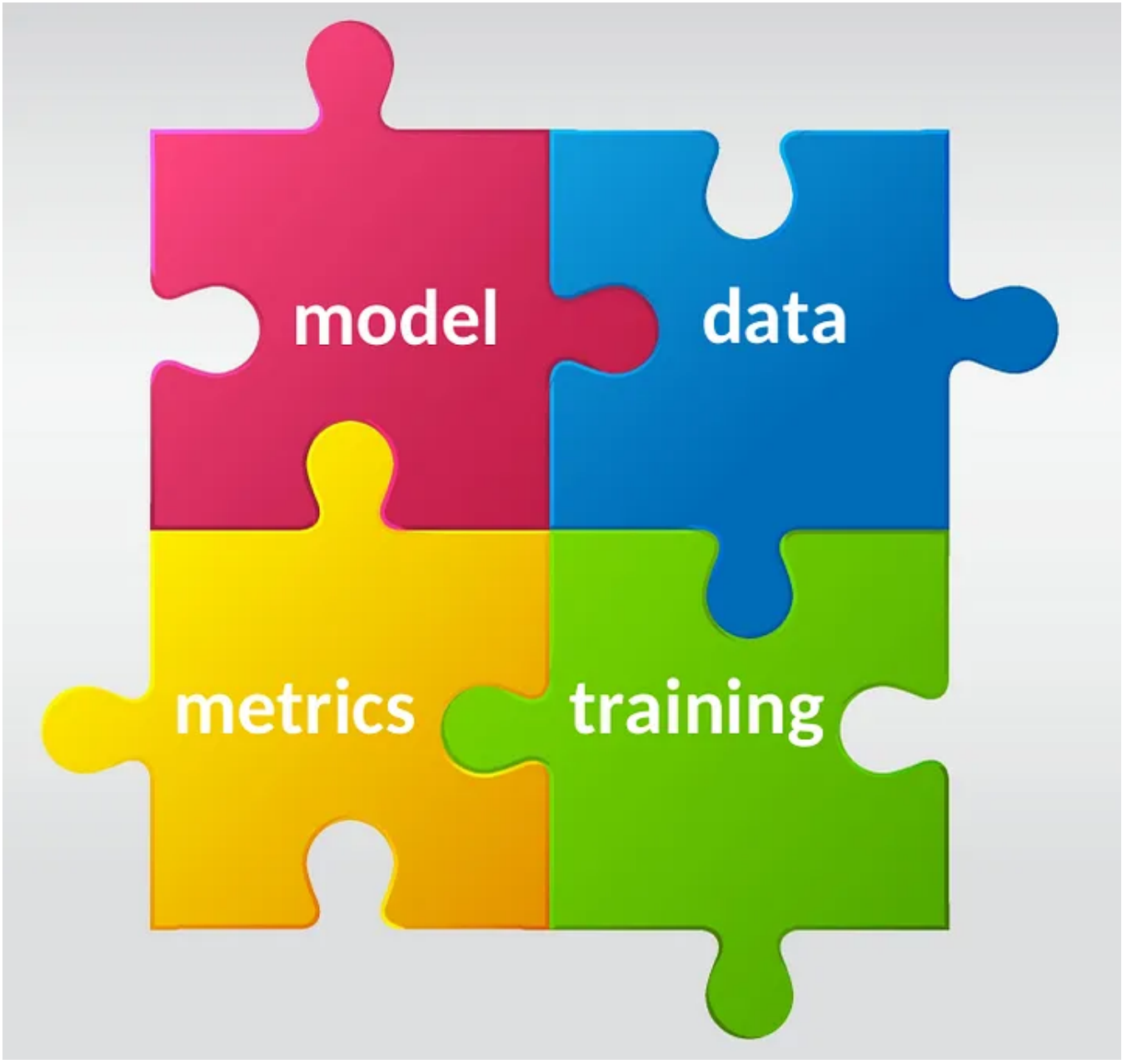
In this story, we will not use one of those high performing off-the-shelf object detectors but develop a new one ourselves, from scratch, using plain python, OpenCV, and Tensorflow.

Why? Well, if you are like me and prefer to learn how things actually work by building them yourself, you probably want to code an object detector from scratch by hand.

> *For the sake of simplicity, our — toy — object detector will be concerned in finding only one single object in each image.*

The Recipe

Roughly speaking, 99% of machine learning projects consist on a simple recipe: define a **model**, get a bunch of **data**, and choose the **metrics** used to **train** and **evaluate** the model.



Most of machine learning projects fit the picture above

Once you define these things, the **training** is a cat-and-mouse game where you need "only" tuning the training hyperparameters in order to achieve the desired performance. This is basically the path in which we are going to walk here.

> *Data augmentation, data preparation, Feature Engineering, etc also play an important role in this game. Covering them is out of scope of this story.*

In the context of our object detector, the model, the data, the metrics and the training are covered in the next sections.

The model

An Object Detection is a combination of two tasks:

- **regression** of the bound-box coordinates
- **classification** of the object label

This means that our model has two outputs: namely the object label and the object bound box. Therefore, the model must combine the tasks of classification and regression.

The Classifier

Let's forget the bound box for a while and begin with a simpler classifier:

```
CLASSES = 2
model = tf.keras.models.Sequential([

    tf.keras.layers.Conv2D(16, kernel_size=3, activation='relu', input_shape=(224, 224, 1)),
    tf.keras.layers.AveragePooling2D(2,2),

    tf.keras.layers.Conv2D(32, kernel_size=3, activation = 'relu'),
    tf.keras.layers.AveragePooling2D(2,2),

    tf.keras.layers.Conv2D(64, kernel_size=3, activation = 'relu'),
    tf.keras.layers.AveragePooling2D(2,2),

    tf.keras.layers.Flatten(),
    tf.keras.layers.Dense(64, activation = 'relu'),

    tf.keras.layers.Dense(CLASSES, activation='softmax')

])
```

medium_building_obj_det_basic_classifier.py hosted with ❤ by GitHub view raw

Defining a classifier using (old-style) sequential

This is a streamline Tensorflow/Keras classifier. It is not able to detect bounding boxes but only the object label. Note that this model has a single input layer and only one output layer. The output layer is set to use Softmax Activation Function as usual in Deep Learning classifiers.

> *Softmax represents the output as a discrete probability distribution which is very useful for represent per class scores.*

Between the input and output layers there are a sequence of internal layers (also called hidden layers). We can rewrite this model using TensorFlow Functional API:

```
CLASSES = 2

def build_classifier(inputs):

    x = tf.keras.layers.Conv2D(16, kernel_size=3, activation='relu', input_shape=(input_size, i
    x = tf.keras.layers.AveragePooling2D(2,2)(x)

    x = tf.keras.layers.Conv2D(32, kernel_size=3, activation = 'relu')(x)
    x = tf.keras.layers.AveragePooling2D(2,2)(x)

    x = tf.keras.layers.Conv2D(64, kernel_size=3, activation = 'relu')(x)
    x = tf.keras.layers.AveragePooling2D(2,2)(x)

    x = tf.keras.layers.Flatten()(inputs)
    x = tf.keras.layers.Dense(64, activation='relu')(x)

    x = tf.keras.layers.Dense(CLASSES, activation='softmax')(x)

    return x
```

medium_building_obj_det_basic_classifier_functional.py hosted with ❤ by GitHub view raw

Defining a classifier using Functional API

This last way to define the model produces the same result in terms of model architecture, size, number of parameters, etc.

> *We are using TensorFlow Functional API here and after because this is more suitable to write complex models like object detectors as we will see soon.*

The Regressor

Now, let's focus on the model to perform the regression task only:

```python
def build_regressor(inputs):

    x = tf.keras.layers.Conv2D(16, kernel_size=3, activation='relu', input_shape=(input_size, i
    x = tf.keras.layers.AveragePooling2D(2,2)(x)

    x = tf.keras.layers.Conv2D(32, kernel_size=3, activation = 'relu')(x)
    x = tf.keras.layers.AveragePooling2D(2,2)(x)

    x = tf.keras.layers.Conv2D(64, kernel_size=3, activation = 'relu')(x)
    x = tf.keras.layers.AveragePooling2D(2,2)(x)

    x = tf.keras.layers.Flatten()(inputs)
    x = tf.keras.layers.Dense(64, activation='relu')(x)

    x = tf.keras.layers.Dense(units = '4')(inputs)(x)

    return x
```

medium_building_obj_det_basic_regressor.py hosted with ❤ by GitHub view raw

Defining a regressor using Functional API

If you check the code carefully, you can note that both regressor and classifier have a similar architecture. Different of the classifier, the last layer in the regressor is set to output a fixed size of 4 values. These 4 values are the 4 bound box coordinates: x, y, width and height.

Now, let's combine the two models into a single one.

The Final Model

Combining the two previous models in a single one is pretty easy using TensorFlow Functional API:

```python
def build_feature_extractor(inputs):

    x = tf.keras.layers.Conv2D(16, kernel_size=3, activation='relu', input_shape=(input_size, i
    x = tf.keras.layers.AveragePooling2D(2,2)(x)

    x = tf.keras.layers.Conv2D(32, kernel_size=3, activation = 'relu')(x)
    x = tf.keras.layers.AveragePooling2D(2,2)(x)

    x = tf.keras.layers.Conv2D(64, kernel_size=3, activation = 'relu')(x)
    x = tf.keras.layers.AveragePooling2D(2,2)(x)

    return x

def build_model_adaptor(inputs):
    x = tf.keras.layers.Flatten()(inputs)
    x = tf.keras.layers.Dense(64, activation='relu')(x)
    return x

def build_classifier_head(inputs):
    return tf.keras.layers.Dense(CLASSES, activation='softmax', name = 'classifier_head')(input

def build_regressor_head(inputs):
    return tf.keras.layers.Dense(units = '4', name = 'regressor_head')(inputs)

def build_model(inputs):

    feature_extractor = build_feature_extractor(inputs)

    model_adaptor = build_model_adaptor(feature_extractor)

    classification_head = build_classifier_head(model_adaptor)

    regressor_head = build_regressor_head(model_adaptor)

    model = tf.keras.Model(inputs = inputs, outputs = [classification_head, regressor_head])

    return model
```

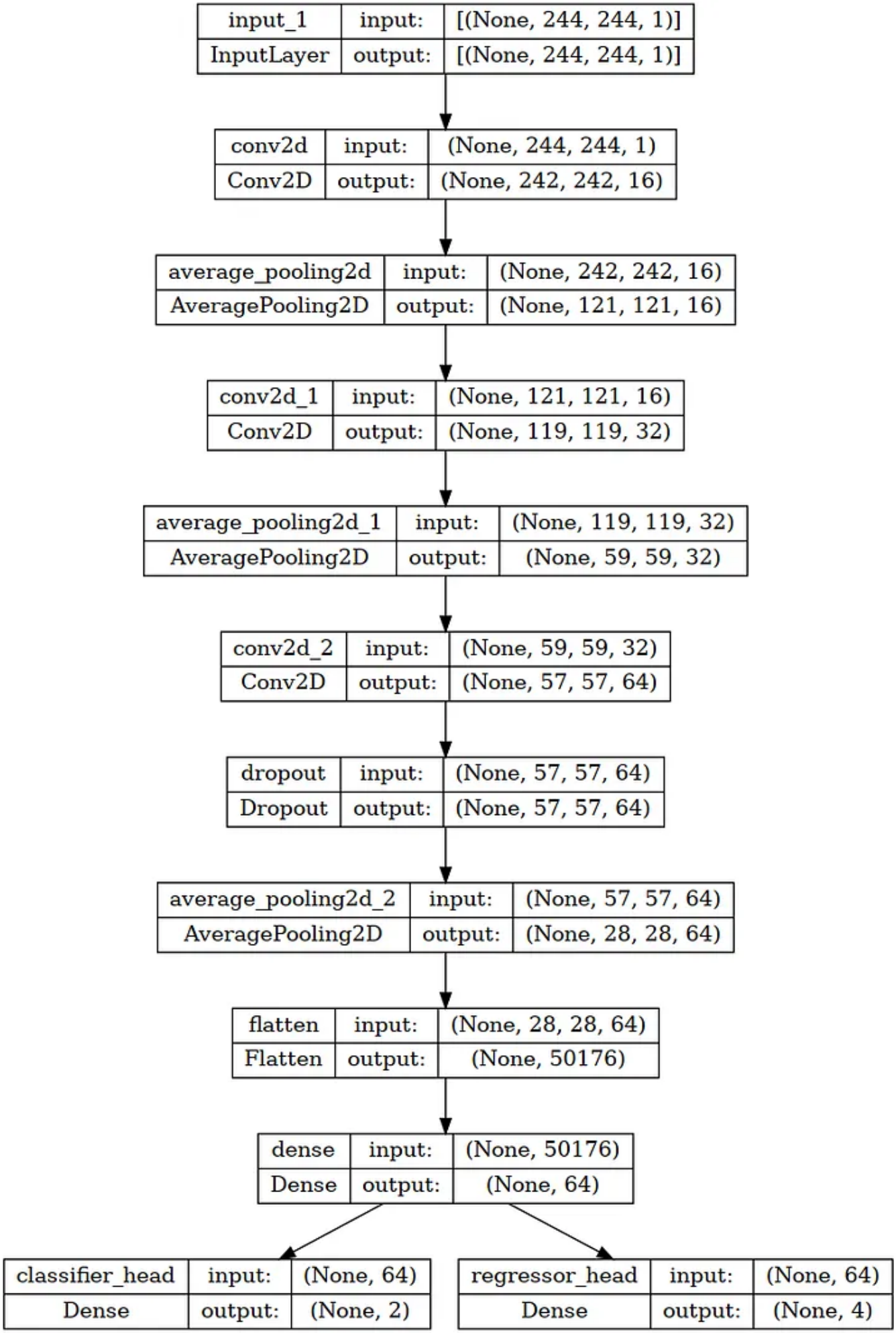
medium_building_obj_det_basic_final_model.py hosted with ❤ by GitHub view raw

Defining the model for our object detector

To keep things tidy, we divided the model code in functions. First, it defines a common structure called “feature extractor” composed by a sequence of convolution layers.

> *In this context, features are something like building blocks of images: low level regularities in the training data.*

The extracted features are feed into the next “adapter” layer which is basically a flatten layer to connect the feature extractor to the two model heads:



The model with two heads: one for classification and one for regression

Note that there are two output branches (or heads): one for the classifier and one for the bound box regressor. During the training, each head will be specialized to its respective task. Now, let's focus on the data.

Getting and preparing the data

Since we are building an object detector from scratch, we cannot use a pre-built model or transfer learning neither. This means that we need to train everything from scratch, starting from the model weights random initialization.

For this mission, we will use the Labeled Mask Dataset. The first thing to do is downloading the data compressed file somewhere on your file system (you need a free kaggle account for this purpose).

Once you have the data, run the following code to prepare the dataset for further usage:

```
import os, random

def list_files(full_data_path = "data/obj/", image_ext = '.jpg', split_percentage = [70, 20]):

    files = []

    discarded = 0
    masked_instance = 0

    for r, d, f in os.walk(full_data_path):
        for file in f:
            if file.endswith(".txt"):

                # first, let's check if there is only one object
                with open(full_data_path + "/" + file, 'r') as fp:
                    lines = fp.readlines()
                    if len(lines) > 1:
                        discarded += 1
                        continue


                strip = file[0:len(file) - len(".txt")]
                # secondly, check if the paired image actually exist
                image_path = full_data_path + "/" + strip + image_ext
                if os.path.isfile(image_path):
                    # checking the class. '0' means masked, '1' for unmasked
                    if lines[0][0] == '0':
                        masked_instance += 1
                    files.append(strip)

    size = len(files)
    print(str(discarded) + " file(s) discarded")
    print(str(size) + " valid case(s)")
    print(str(masked_instance) + " are masked cases")

    random.shuffle(files)

    split_training = int(split_percentage[0] * size / 100)
    split_validation = split_training + int(split_percentage[1] * size / 100)

    return files[0:split_training], files[split_training:split_validation], files[split_validat

training_files, validation_files, test_files = list_files()

print(str(len(training_files)) + " training files")
print(str(len(validation_files)) + " validation files")
print(str(len(test_files)) + " test files")
```

medium_building_obj_det_prep_data.py hosted with ❤ by GitHub view raw

This code basically generates three lists: one for training (holding 70% of the data), one for validation (20% of the data) and one for test (the last 10%). The code also shuffle the data in order to avoid any natural bias:

```
training_files, validation_files, test_files = list_files()

print(str(len(training_files)) + " training files")
print(str(len(validation_files)) + " validation files")
print(str(len(test_files)) + " test files")
```

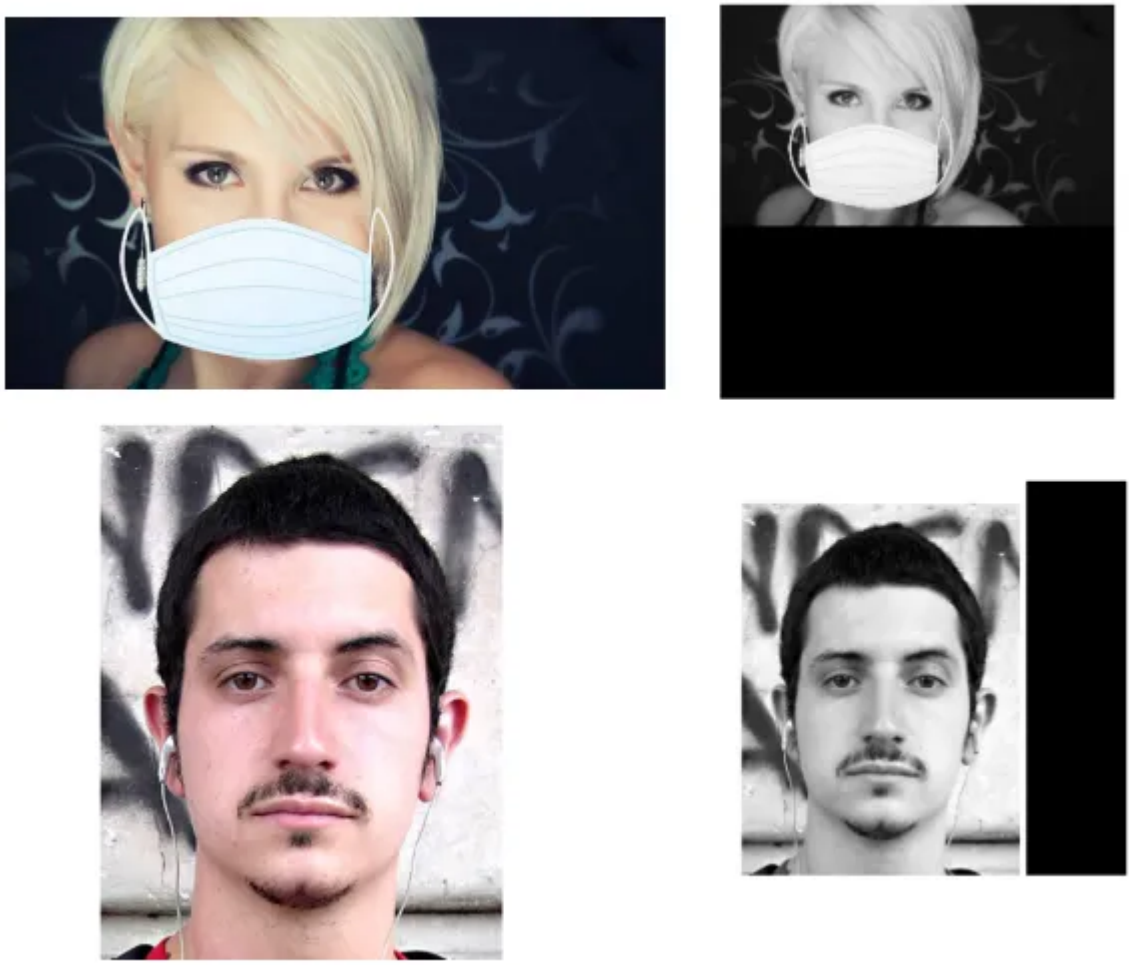
[6] ✓ 0.0s

```
218 file(s) discarded
1292 valid case(s)
832 are masked cases
904 training files
258 validation files
130 test files
```

> *Note that we are removing the images with more than one masks/objects. This is because we are building a simple object detector able to detect only a single object in an image.*

Formatting data

Since our model has a fixed 244 x 244 input layer, we need to format any input image before feed it to the model (to train or to predict). For example, the images below are formatted to fit a 244 x 244 black frame:



Formatting images to further usage

The code to format the images is shown below:

```python
input_size = 244

def format_image(img, box):
    height, width = img.shape
    max_size = max(height, width)
    r = max_size / input_size
    new_width = int(width / r)
    new_height = int(height / r)
    new_size = (new_width, new_height)
    resized = cv.resize(img, new_size, interpolation= cv.INTER_LINEAR)
    new_image = np.zeros((input_size, input_size), dtype=np.uint8)
    new_image[0:new_height, 0:new_width] = resized

    x, y, w, h = box[0], box[1], box[2], box[3]
    new_box = [int((x - 0.5*w)* width / r), int((y - 0.5*h) * height / r), int(w*width / r), in

    return new_image, new_box
```

medium_building_obj_det_format_img.py hosted with ❤ by GitHub view raw

Note that the bounding box must be formatted as well in order to take the new image dimensions in consideration. Let's perform a simple sanity check:

```python
plt.figure(figsize=(20, 10))
for images, labels in train_ds.take(1):
    for i in range(BATCH_SIZE):
        ax = plt.subplot(4, BATCH_SIZE//4, i + 1)
        label = labels[0][i]
        box = (labels[1][i] * input_size)
        box = tf.cast(box, tf.int32)

        image = images[i].numpy().astype("float") * 255.0
        image = image.astype(np.uint8)
        image_color = cv.cvtColor(image, cv.COLOR_GRAY2RGB)

        color = (0, 0, 255)
        if label[0] > 0:
            color = (0, 255, 0)

        cv.rectangle(image_color, box.numpy(), color, 2)

        plt.imshow(image_color)
        plt.axis("off")
```

medium_building_obj_det_checking_data.py hosted with ❤ by GitHub view raw

Checking the training data after formatting images & bounding boxes

Indeed, both images and respective annotations are correctly formatted.

Getting the Metrics and Loss Functions

Since our model must implement two tasks — classification and regression — we need two different Loss Functions:

- One for the classification task: we may use any Loss Function usually found in only-classification tasks like Categorical Crossentropy.
- One for the bound box regression: we can use a regression Loss Function such as Mean Squared Error.

Finally, we have all the required ingredients to get our model working. The next step is training the model to obtain our desired Object Detector!

Training the Model

At this point, I assume that you are familiar with machine learning basics, like training and validation sets, epochs, learning rate, optimizers and so on. Almost everything now can be taken as a hyperparameter and I really recommend you to play with them:

```
1
2  model = build_model(tf.keras.layers.Input(shape=(input_size, input_size, 1,)))
3
4  model.compile(optimizer=tf.keras.optimizers.Adam(),
5      loss = {'classifier_head' : 'categorical_crossentropy', 'regressor_head' : 'mse' },
6      metrics = {'classifier_head' : 'accuracy', 'regressor_head' : 'mse' })
7
8  EPOCHS = 100
9  BATCH_SIZE = 32
10
11 history = model.fit(train_ds,
12                     steps_per_epoch=(len(training_files) // BATCH_SIZE),
13                     validation_data=validation_ds, validation_steps=1,
14                     epochs=EPOCHS)
```

medium_building_obj_det_training_model.py hosted with ❤ by GitHub view raw

training our object detector model

This code outputs something like:

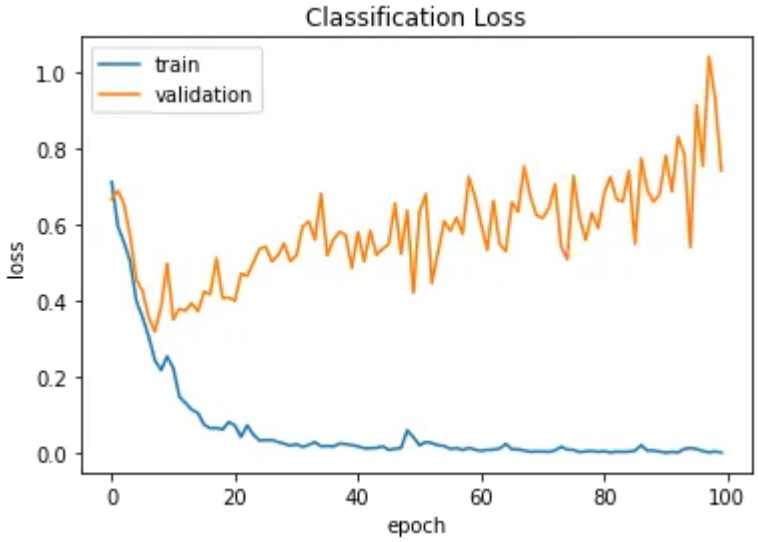
```
EPOCHS = 100
history = model.fit(train_ds,
                    steps_per_epoch=(len(training_files) // BATCH_SIZE),
                    validation_data=validation_ds, validation_steps=1,
                    epochs=EPOCHS)
[24]  1m 49.4s
...  Output exceeds the size limit. Open the full output data in a text editor
Epoch 1/100
28/28 [==============================] - 7s 63ms/step - loss: 0.7395 - classifier_head_loss: 0.63
Epoch 2/100
28/28 [==============================] - 1s 39ms/step - loss: 0.5999 - classifier_head_loss: 0.57
Epoch 3/100
28/28 [==============================] - 1s 39ms/step - loss: 0.5599 - classifier_head_loss: 0.53
Epoch 4/100
28/28 [==============================] - 1s 39ms/step - loss: 0.4619 - classifier_head_loss: 0.44
Epoch 5/100
28/28 [==============================] - 1s 39ms/step - loss: 0.3859 - classifier_head_loss: 0.36
```

> *If you get memory problems when running this train, check the GPU usage and reduce the `BATCH_SIZE` for a suitable value.*

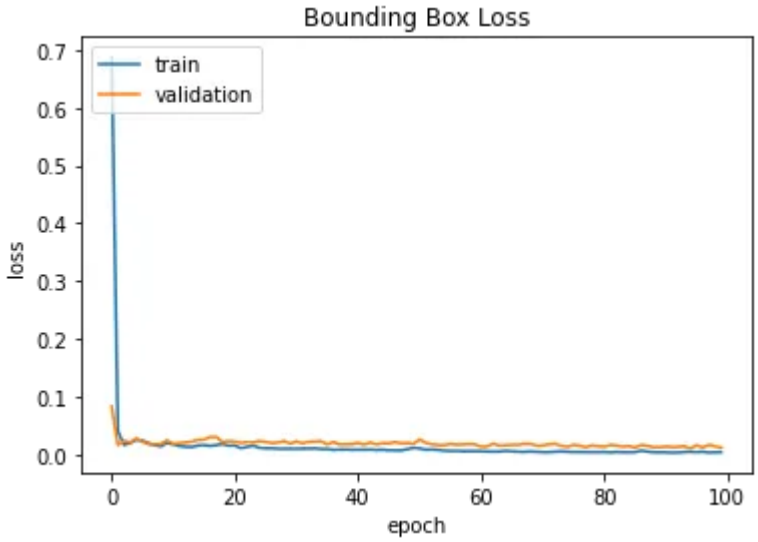
Depending on your environment, your hardware, the presence or absence of GPU or TPU, the training can take last than one minute or hours to finish. Stay frost and once the training finishes move to the next step.

Checking the Training Performance

After the training finishes, we can check how the Loss Function performed during the process. Remember: we have two losses!



Classification loss over the epochs



Regression loss over the epochs

A challenge in training hybrid models like this one is that the model can overfit one output but not the other =D It seems to be the case here: the classification loss clearly shows an system in overfitting but the bounding box loss doesn't. We will discuss more about overfitting later on.

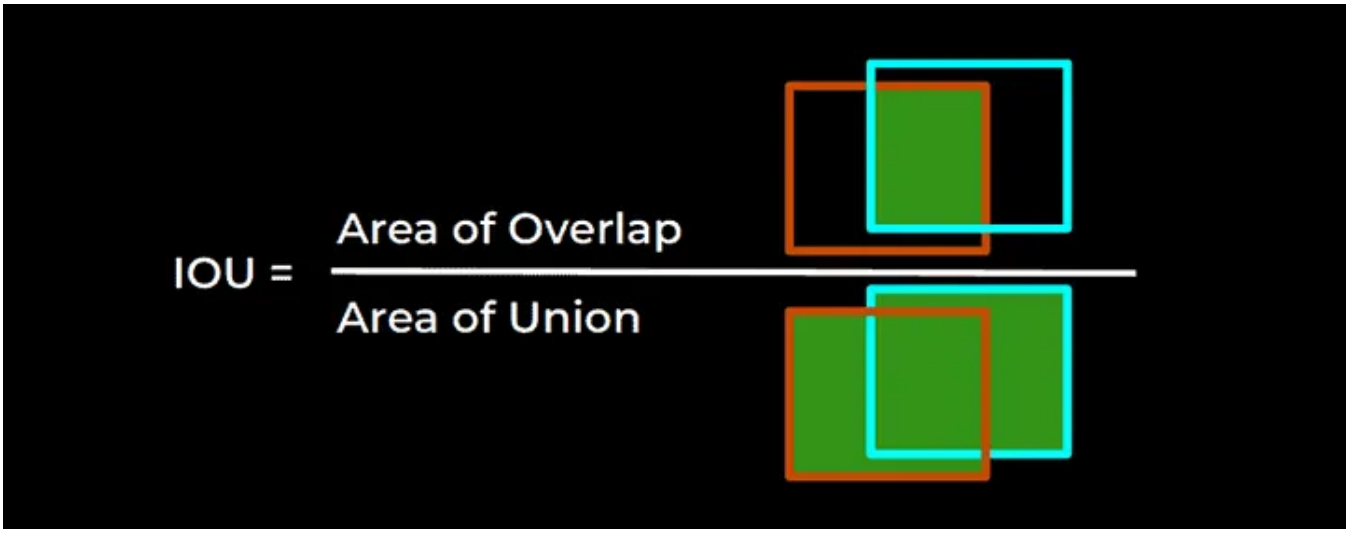
Model Selection and Evaluation

Each time you run `model.fit(...)` you end up with a different model with a different performance. It turns out that we need to decide which model is the best for further steps. The process of making this choice is called **Model Selection.**

In model selection, we compare the model performance using one or more metrics. Metrics are similar to Loss Functions but not exactly the same. One popular metric for object detectors is **IoU — Intersection over Union**, describe below.

Intersection over Union

IoU scores how well the predicted bound box overlaps the actual bound box. The idea behind IoU is pretty simple: compare the intersection and union areas between the predicted and actual bound boxes by dividing the intersection by the union, as shown in the following image:



source: learn opencv

IoU provides a higher score always when the predicted bounding box best matches the actual bounding box (also called ground-truth):

source: pyimagesearch

We can realize that IoU ranges from 0 up to 1. The following code was adapted from an article on pyimagesearch website:

```python
# adapted from: https://pyimagesearch.com/2016/11/07/intersection-over-union-iou-for-object-det
def intersection_over_union(boxA, boxB):
        xA = max(boxA[0], boxB[0])
        yA = max(boxA[1], boxB[1])
        xB = min(boxA[0] + boxA[2], boxB[0] + boxB[2])
        yB = min(boxA[1] + boxA[3], boxB[1] + boxB[3])
        interArea = max(0, xB - xA + 1) * max(0, yB - yA + 1)
        boxAArea = (boxA[2] + 1) * (boxA[3] + 1)
        boxBArea = (boxB[2] + 1) * (boxB[3] + 1)
        iou = interArea / float(boxAArea + boxBArea - interArea)
        return iou
```

medium_building_obj_det_iou.py hosted with ❤ by GitHub view raw

Now, we can run IoU over our **test set:**

```
plt.figure(figsize=(12, 10))

test_list = list(test_ds.take(20).as_numpy_iterator())

image, labels = test_list[0]

for i in range(len(test_list)):

    ax = plt.subplot(4, 5, i + 1)
    image, labels = test_list[i]

    predictions = model(image)

    predicted_box = predictions[1][0] * input_size
    predicted_box = tf.cast(predicted_box, tf.int32)

    predicted_label = predictions[0][0]

    image = image[0]

    actual_label = labels[0][0]
    actual_box = labels[1][0] * input_size
    actual_box = tf.cast(actual_box, tf.int32)

    image = image.astype("float") * 255.0
    image = image.astype(np.uint8)
    image_color = cv.cvtColor(image, cv.COLOR_GRAY2RGB)

    color = (255, 0, 0)
    # print box red if predicted and actual label do not match
    if (predicted_label[0] > 0.5 and actual_label[0] > 0) or (predicted_label[0] < 0.5 and actu
        color = (0, 255, 0)

    img_label = "unmasked"
    if predicted_label[0] > 0.5:
        img_label = "masked"

    predicted_box_n = predicted_box.numpy()
    cv.rectangle(image_color, predicted_box_n, color, 2)
    cv.rectangle(image_color, actual_box.numpy(), (0, 0, 255), 2)
    cv.rectangle(image_color, (predicted_box_n[0], predicted_box_n[1] + predicted_box_n[3] - 20
    cv.putText(image_color, img_label, (predicted_box_n[0] + 5, predicted_box_n[1] + predicted_

    IoU = intersection_over_union(predicted_box.numpy(), actual_box.numpy())

    plt.title("IoU:" + format(IoU, '.4f'))
    plt.imshow(image_color)
    plt.axis("off")
```

medium_building_obj_det_running_iou.py hosted with ❤ by GitHub view raw

Predictions on test dataset

> *Always when we evaluate a model, we need to take care to perform our analysis on "new data" or "unseen data", i.e, data not used during the model training.*

The green/red rectangles represent the predicted bound boxes. A green bound box means that the predicted and actual label match. A red rectangle means that the model was not able to correctly classify that instance.

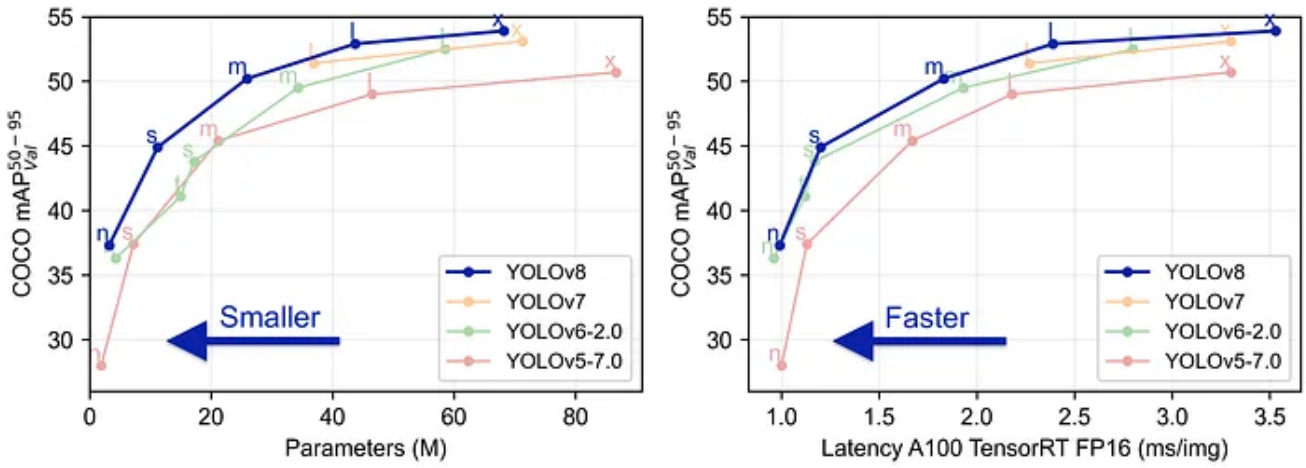
Very often we define a threshold α and then we calculate TP, TN, FP, FN, recall, precision, etc based on how many times IoU exceeds α.

$$\text{Precision} = \frac{tp}{tp + fp}$$

$$\text{Recall} = \frac{tp}{tp + fn}$$

source: wikipedia

And, using precision and recall, we can obtain a key indicator for object detectors: Mean Average Precision, or simply **mAP** .



Comparison between different YOLO version using mAP (source)

Indeed, mAP is the most used indicator when we are comparing different object detectors.

> *The details behind mAP and its implementation are beyond the scope of this story. This article on roboflow does a great job in explaining this: https://blog.roboflow.com/mean-average-precision/*

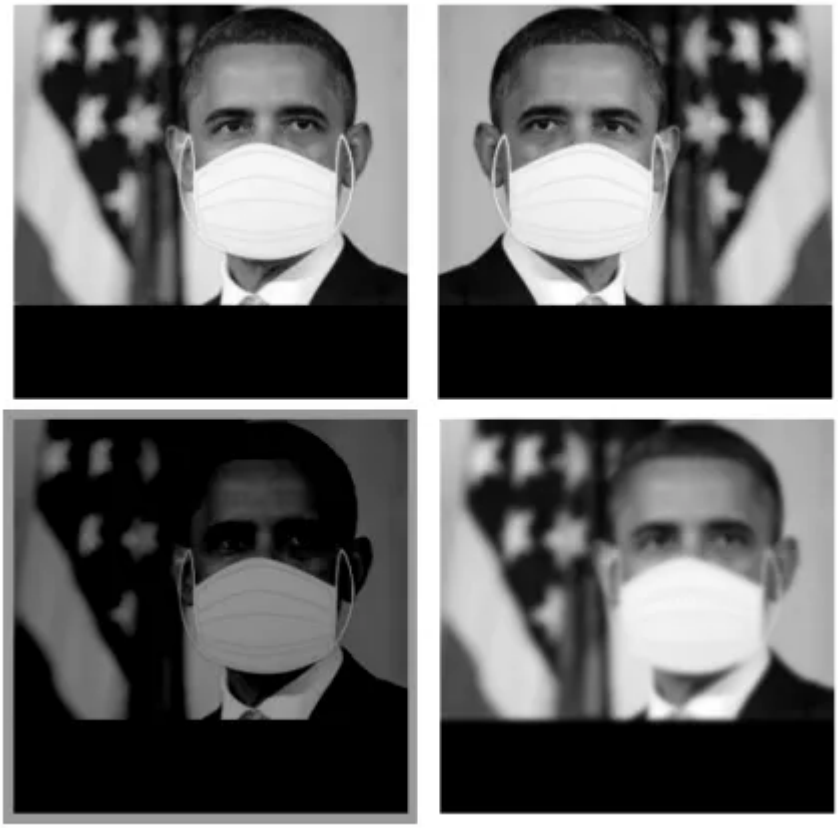
Getting Better Results

Given the small size of training data (only 904 training instances!) and model (only 3,235,014 parameters!) the achieved performance was surprisingly good. But we can achieve better results with a little bit of effort.

more samples from the test set

First, a larger training dataset will improve the results significantly. It turns out that using only a few training examples makes the model prone to overfitting. Indeed, using large datasets is a key factor when we are building a complex, production-oriented model.

One simple way to get more data is data augmentation. Using built-in OpenCV or Tensorflow functions, we can easily generate new data by rotating, translating, adding noise, changing brightness, etc.



horizontal flip, changing brightness and blurring in action

In practice, data augmentation is part of any pipeline in real world object detectors.

> *Make sure that your augmented data do not accidentally end up in the test or validation sets!*

Once we have more data, we can increase the model capacity by adding more layers. In addition, tweaking with model features, like activation functions and pooling types, also worth a lot.

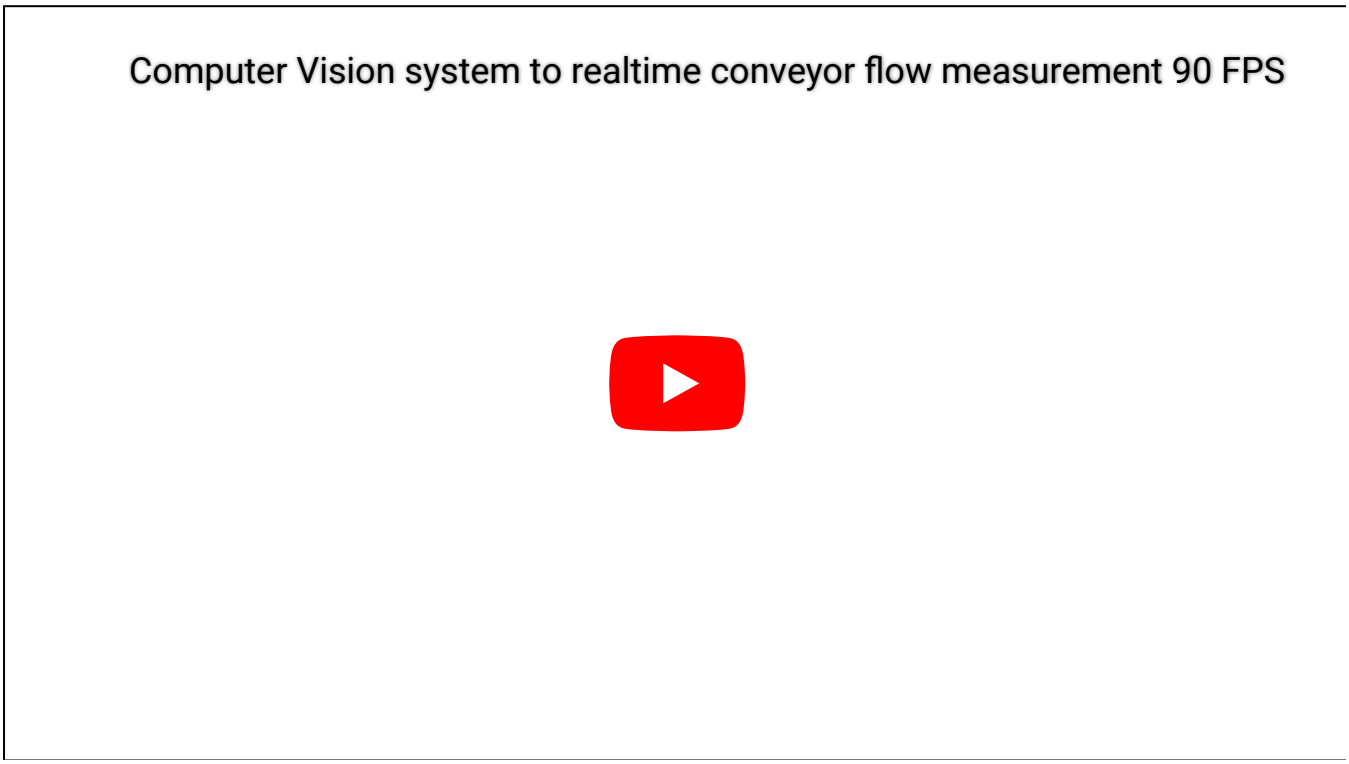
> *Adding layers and trainable parameters to the model make it more complex. But if your model is too complex and you have only a few data to train, you probably will end up with an overfitted, useless model at hands.*

Finally, we can try to change the training, for example, by changing the optimizer or playing with the hyperparameters.

Conclusions

In this story, we covered the basics of building a toy object detector from scratch using TensorFlow. We aimed to illustrate the challenges and concepts behind Object Detectors providing some understanding of how this technology works.

In my professional life, in 2014–2017, I could work in a project to develop a real time object detector for production purposes. That time, tensorflow/pytorch and the Deep Learning technology were not ready yet. I remember how so hard was to achieve a working solution developing our real time tracking system in pure C++ without even tensorflow or pytorch. If you are curious about this project, you can check the 2017 video below:



An ancient real time object detector in production in 2017

Today, state-of-art object detectors like YOLOv5 or YOLOv8 are way more powerful if compared to the toy implementation shown here. They perform multiscale and multiple objects detection extremely fast even on CPUs.

> *Our advice is using YOLO in real world applications always as need. The model shown here is for educational purposes only.*

Source Code

You can check the full code on this colab notebook . If the notebook is not available for any reason, try this repository on github.

References

Tensorflow Funcional API: https://www.tensorflow.org/guide/keras/functional

Intersection over Union: https://pyimagesearch.com/2016/11/07/intersection-over-union-iou-for-object-detection/

Labeled Mask dataset: https://www.kaggle.com/datasets/techzizou/labeled-mask-dataset-yolo-darknet

YOLO v5/v8: https://ultralytics.com/

Mean Average Precision: https://blog.roboflow.com/mean-average-precision/

BECOME a WRITER at MLearning.ai

Mlearning.ai Submission Suggestions
How to become a writer on Mlearning.ai
medium.com

Deep Learning | Object Detection | TensorFlow | Ml So Good

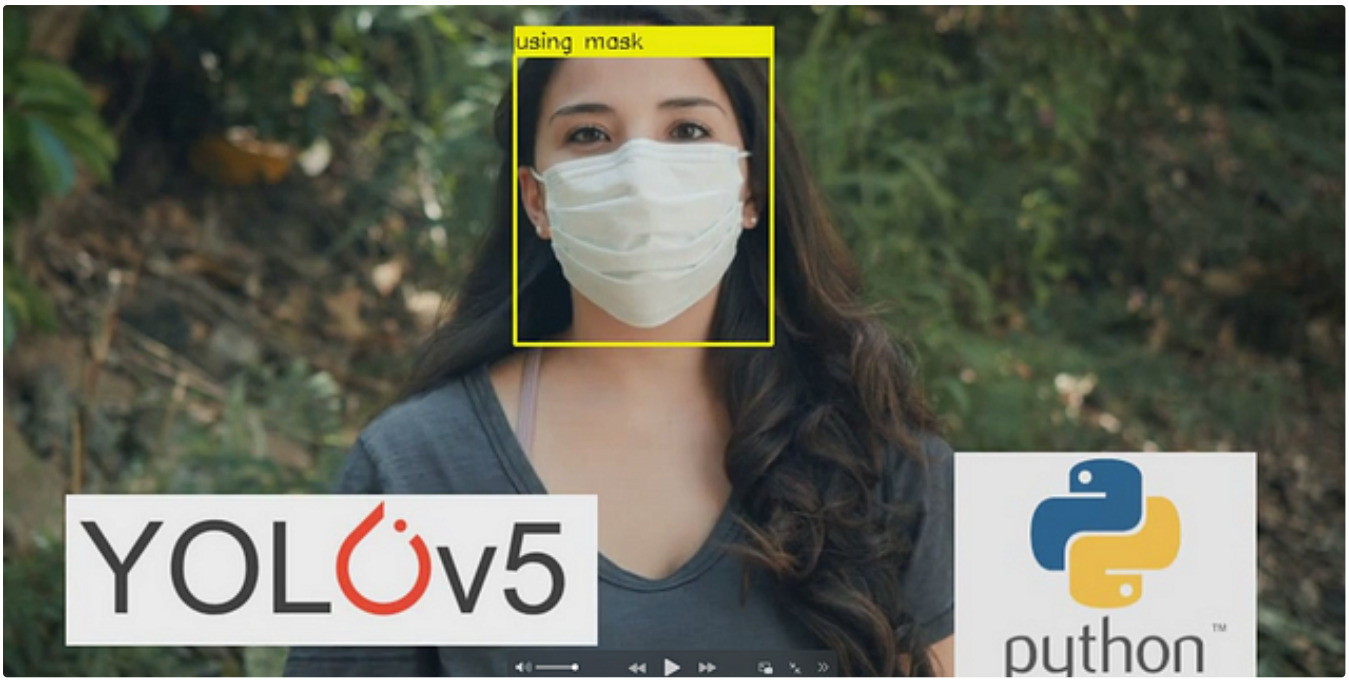
Follow

Written by Luiz doleron

361 Followers · Writer for MLearning.ai

Artificial Intelligence, Computer Vision & System Architect https://github.com/doleron https://www.linkedin.com/in/doleron/

More from Luiz doleron and MLearning.ai



Luiz doleron in MLearning.ai

Training YOLOv5 custom dataset with ease

YOLOv5 is one of the most high-performing object detector out there. It is fast, has high accuracy and is incredibly easy to train.

6 min read · Jan 26, 2022

225 7

Yennhi95zz in MLearning.ai

How to Earn Money from Your Python Coding Skills: 10 Monetization Projects Ideas

Make the most of your learning abilities with Python and gain an edge

✨ · 13 min read · Nov 26

519 6

Maximilian Vogel in MLearning.ai

The ChatGPT list of lists: A collection of 3000+ prompts, examples, use-cases, tools, APIs...

Updated Oct-18, 2023. New developer resources, marketing & SEO prompts, new prompt engineering courses, masterclasses and tutorials.

11 min read · Feb 8

11.1K 135

Luiz doleron in MLearning.ai

Detecting objects with YOLOv5, OpenCV, Python and C++

It is not uncommon people to think of computer vision as a hard subject to understand and, also, hard to get running. Indeed, not long ago...

6 min read · Jan 23, 2022

 335  5 

See all from Luiz doleron

See all from MLearning.ai

Recommended from Medium

Everton Gomede, PhD

RetinaNet: Advancing Object Detection in Computer Vision

Introduction

6 min read · Aug 27

2

Alex

Hosting YOLOv8 With FastAPI

Introduction

8 min read · Nov 1

18

Lists

Practical Guides to Machine Learning

10 stories · 836 saves

Natural Language Processing

1036 stories · 512 saves

data science and AI

38 stories · 22 saves

Tech & Tools

15 stories · 115 saves

Yash Raj Mani in Python in Plain English

Object Detection with YOLO and OpenCV from Video Frames

In this article, we will explore how to build a simple object detection project using YOLO (You Only Look Once) and OpenCV - PYTHON.

5 min read · Oct 12

19

Elven Kim

How to convert Object segmentation in YOLOv8 custom model to TFlite

Have you use Object Detection before? And frustrated by the bounding boxes which did not enclose object properly? Or worse still, it gives...

5 min read · Sep 11

12

Mohit Gupta

Computer Vision Blogs: Object Detection with Yolov8—Train your own custom object detection model

If you've found your way here, chances are you're struggling with the intricacies of object detection. I promise to you that by the end of...

8 min read · Aug 11

8 1

Rahil Moosavi

Counting People On Escalator Using Yolov8 and OpenCV from scratch

Object detection is one of the important phenomena in the field of computer vision. On the other hand, computer vision is progressing with...

5 min read · Sep 16

 4 

See more recommendations