**НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ УКРАЇНИ**

**«КИЇВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ  
імені Ігоря Сікорського»**

**Факультет прикладної математики**

**Кафедра прикладної математики**

«До захисту допущено»

Завідувач кафедри

\_\_\_\_\_\_\_\_\_\_\_\_\_ Олег ЧЕРТОВ

«\_\_\_» \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ 2021 р.

**Дипломна робота**

**на здобуття ступеня бакалавра**

**за освітньо-професійною програмою «Наука про дані та математичне моделювання»**

**спеціальності 113 «Прикладна математика»**

**на тему: «Математичне та програмне забезпечення для системи приховання облич та реклами на відео у реальному часі»**

Виконав (-ла):

студент (-ка) IV курсу, групи КМ-73

Тищенко Богдан Юрійович \_\_\_\_\_\_\_\_\_\_

Керівник:

Доцент, кандидат наук,

Олефір Олександр Степанович \_\_\_\_\_\_\_\_\_\_

Консультант з нормконтролю:

Старший викладач,

Мальчиков Володимир Вікторович \_\_\_\_\_\_\_\_\_\_

Рецензент:

Старший викладач,

Дробязко Ірина Павлівна \_\_\_\_\_\_\_\_\_\_

Засвідчую, що в цій дипломній роботі немає запозичень із праць інших авторів без відповідних посилань.

Студент \_\_\_\_\_\_\_\_\_\_\_\_

Київ — 2021 року**Національний технічний університет України**

**«Київський політехнічний інститут імені Ігоря Сікорського»**

**Факультет прикладної математики**

**Кафедра прикладної математики**

Рівень вищої освіти – перший (бакалаврський)

Спеціальність – 113 «Прикладна математика»

Освітньо-професійна програма «Наука про дані та математичне моделювання»

ЗАТВЕРДЖУЮ

Завідувач кафедри

\_\_\_\_\_\_\_\_\_\_\_\_\_ Олег ЧЕРТОВ

«\_\_\_» \_\_\_\_\_\_\_\_\_\_\_\_\_\_ 2021 р.

**ЗАВДАННЯ**

**на дипломну роботу студенту**

Тищенко Богдану Юрійовичу

1. Тема роботи: «Математичне та програмне забезпечення для системи приховання облич та реклами на відео у реальному часі», керівник роботи Олефір Олександр Степанович, канд. техн. наук, доцент, затверджені наказом по університету від «\_» \_\_\_\_\_\_ 2021 р. № \_\_\_\_-С.

2. Термін подання студентом роботи: «9» червня 2021 р.

3. Вихідні дані до роботи: розроблювана система повинна видавати оброблений відео потік зі швидкістю >24 кадри на секунду, мати точність детекції реклами та облич >60% (mAP50).

4. Зміст роботи: виконати аналіз існуючих методів розв’язання задачі, вибрати методи для детекції облич та реклами на зображенні, виконати навчання вибраних моделей, спроектувати автоматизовану систему детекції облич та реклами, здійснити програмну реалізацію розробленої системи, провести тестування розробленої системи.

5. Перелік ілюстративного матеріалу: приклади навчальних даних, архітектурні графи нейронних мереж, блок-схеми розроблених алгоритмів, схема взаємодії модулів системи, графіки прогресу тренування моделей, знімки екранних форм.

6. Консультанти розділів роботи:

|  |  |  |  |
| --- | --- | --- | --- |
| Розділ | Прізвище, ініціали та посада консультанта | Підпис, дата | |
| завдання видав | завдання прийняв |
| Розділ 3. Математичне забезпечення | Мальчиков В. В., старший викладач |  |  |

7. Дата видачі завдання: «01» лютого 2021 р.

Календарний план

| № з/п | Назва етапів виконання дипломної роботи | Термін виконання етапів роботи | Примітка |
| --- | --- | --- | --- |
| 1 | Огляд літератури за тематикою та збір даних | 1.03.2021 |  |
| 2 | Проведення порівняльного аналізу методів детекції | 1.4.2021 |  |
| 3 | Написання теоретичної частини роботи, описання використовуваних методів | 26.04.2021 |  |
| 4 | Розробка математичної моделі для задачі приховання облич та реклами у реальному часі | 02.05.2021 |  |
| 5 | Тестування та відлагодження математичної моделі й методів, оцінка точності | 10.05.2021 |  |
| 6 | Імплементація отриманих результатів у програмний продукт | 17.05.2021 |  |
| 7 | Висновки та оформлення пояснювальної записки | 01.06.2021 |  |

|  |  |  |
| --- | --- | --- |
| Студент | \_\_\_\_\_\_\_\_\_\_\_\_\_\_ | Богдан ТИЩЕНКО |
| Керівник роботи | \_\_\_\_\_\_\_\_\_\_\_\_\_\_ | Олександр ОЛЕФІР |

Анотація

Даний звіт присвячено результатам проходження переддипломної практики на тему «Створення математичного та програмного забезпечення для системи приховання облич та реклами у реальному часі», яка відбувалася з 12 квітня по 16 травня 2021 року на базі моєї квартири. У рамках практики поставлено задачу на дипломне проектування, сформульовано критерії вибору методу розв’язання поставленої задачі.

Розглянуто ряд методів для детекції об’єктів: алгоритм Віола-Джонса та методів побудованих на основі згорткових нейронних мереж: Мask-RCNN, MTCNN та методи родини YOLO: YOLO, YOLOv3, YOLOv5, YOLOFace, YOLACT. На основі сформульованих критеріїв для розв’язання поставленої задачі вибрано метод MТCNN для детекції облич та метод YOLOv5 для детекції рекламних банерів.

Abstract

The thesis is presented in 69 pages. It contains 2 appendixes and bibliography of 16 references. 29 figures and 11 tables are given in the thesis.

The purpose of this thesis is to develop mathematical model and software for the real-time face and advertising hiding system.

The functional and non-functional requirements for our end product have been formulated. Summarizing, the program should run at 24+ fps on Nvidia GeForce 1650 GPU, it should be able to accurately detect faces and advertising in incoming images and it should have a graphical user interface.

When analyzing the existing solutions of certain problems, namely methods of object detection: Mask-RCNN, MTCNN, Violi-Jones algorithm and methodical methods YOLO: YOLO, YOLOv3, YOLOv5, YOLOFace, YOLACT. Their comparison with the review in terms of speed and accuracy of detection, as well as taking into account their algorithmic features. Based on the formulated arteries for linking the task of the selected MTCNN method for shape detection and the YOLOv5 method for detection of banner ads.

When analyzing the existing solutions, we found another optimal methodology variant we could have used. This variant is to use a single unified YOLO model for both face and advertising detection. This one would give much higher fps and training speed while having a slight decrease in face detection accuracy.

The model description part of the thesis tries to explain and mathematically define the following concepts: artificial neural networks, convolutional neural network and its structural parts, non-maximal suppression, intersection over union, YOLO model and its structural parts, dataset augmentation methods, MTCNN model.

A system has been developed that implements the selected methods. The system consists of two parts: training the model and using trained models.

YOLOv5 model has been trained using the advertising banners dataset. The loss value was being analyzed during the training. The trained model was evaluated on the validation sample. The following accuracy metrics have been analyzed: recall, precision and mean average precision (mAP). As a result of our training the model had the accuracy of 70% mAP.

A software for desktop was developed. Created program is able to process images and webcam stream by detecting faces and advertising banners. The program is capable of streaming processed webcam video in real-time outputting 24 frames per second.

The user interface was developed. It can process webcam video stream and individual pictures, record video frames from the webcam.

Keywords: object detection, convolutional neural networks, YOLO, dataset augmentation, Python, augmented reality, real-time.

ЗМІСТ

[List of abbreviations 8](#_Toc74056161)

[Motivation 9](#_Toc74056162)

[1 Task description 10](#_Toc74056163)

[2 The analysis of existing methods and datasets 11](#_Toc74056164)

[2.0.1 On non-machine learning based approaches 11](#_Toc74056165)

[2.0.2 Obtaining GPU results 11](#_Toc74056166)

[2.1 Datasets 12](#_Toc74056167)

[2.1.1 Advertising dataset 12](#_Toc74056168)

[2.1.2 Faces dataset 12](#_Toc74056169)

[2.2 Advertising detection 13](#_Toc74056170)

[2.2.1 Mask R-CNN 13](#_Toc74056171)

[2.2.2 YOLACT 14](#_Toc74056172)

[2.2.3 YOLO 14](#_Toc74056173)

[2.2.4 YOLOv3 15](#_Toc74056174)

[2.2.5 YOLOv5 16](#_Toc74056175)

[2.3 Face detection 17](#_Toc74056176)

[2.3.1 MTCNN 17](#_Toc74056177)

[2.3.2 YOLO 17](#_Toc74056178)

[2.3.3 Viola-Jones method 18](#_Toc74056179)

[2.4 Comparing methods 18](#_Toc74056180)

[2.4.1 For advertising 18](#_Toc74056181)

[2.4.2 For faces 19](#_Toc74056182)

[2.5 Conclusions 20](#_Toc74056183)

[3 Methodology description 21](#_Toc74056184)

[3.1 Neural network 21](#_Toc74056185)

[3.2 Training 22](#_Toc74056186)

[3.3 Convolutional neural network 23](#_Toc74056187)

[3.3.1 Convolutional layer 24](#_Toc74056188)

[3.3.2 Pooling layer 25](#_Toc74056189)

[3.3.3 Dropout layer 25](#_Toc74056190)

[3.4 Intersection Over Union 26](#_Toc74056191)

[3.5 Non-max suppression 27](#_Toc74056192)

[3.6 YOLO 27](#_Toc74056193)

[3.6.1 CNN architecture 27](#_Toc74056194)

[3.6.2 Algorithm 28](#_Toc74056195)

[3.6.3 Anchor boxes 29](#_Toc74056196)

[3.6.4 Training 30](#_Toc74056197)

[3.7 Measuring accuracy 32](#_Toc74056198)

[3.8 Data augmentation 33](#_Toc74056199)

[3.8.1 Distortion 34](#_Toc74056200)

[3.8.2 Image occlusion 34](#_Toc74056201)

[3.8.3 Augmentations for YOLO 36](#_Toc74056202)

[3.9 Face detection 37](#_Toc74056203)

[3.10 MTCNN 38](#_Toc74056204)

[3.10.1 Algorithm 39](#_Toc74056205)

[3.10.2 Training 40](#_Toc74056206)

[4 Developed software 42](#_Toc74056207)

[4.1 Structural model 42](#_Toc74056208)

[4.1.1 Training 42](#_Toc74056209)

[4.1.2 Wrapping model into application 44](#_Toc74056210)

[4.2 Dependencies 45](#_Toc74056211)

[4.3 Evaluation 46](#_Toc74056212)

[4.3.1 Training YOLO 46](#_Toc74056213)

[4.3.2 Validation 47](#_Toc74056214)

[4.4 Experimental results 48](#_Toc74056215)

[5 Future work 53](#_Toc74056216)

[6 Сonclusions 54](#_Toc74056217)

[References 55](#_Toc74056218)

[Appendix А. Code listing. 57](#_Toc74056219)

# List of abbreviations

MTCNN – Multi-Task Convolutional Neural Network.

YOLO – You Only Look Once.

YOLACT – You Only Look aT CoefficienTs.

RCNN – Residual Convolutional Neural Network.

CNN – Convolutional Neural Network.

GPU – Graphical Processing Unit.

IOU – Intersection Over Union.

SOTA – State-Of-The-Art.

NN – Neural Network.

FCNN – Fully Convolutional Neural Network.

GUI – Graphical User Interface.

CAM – Class Activation Maps.

# Motivation

As both the augmented reality and image processing technologies become more and more advanced and ubiquitous we can finally start to think about solving tasks that require merging them together. This kind of combination will further help in making great tools for humans to become happier, hopefully. An example of such tool can be one that provides an environment for human to rest, by reducing the amount of distractions.

It is known that modern human has been around for millions of years. They relied on their pack the size of which was gradually increasing from 10 to 200 people over those years. It is natural for a human to be around the same circle of people through their whole life. Back then, and not too long ago, everyone in the group knew each other. If you meet someone - this is an opportunity to improve your relationship. Behave yourself well and try to be friendly, because in cooperation is the power of the "naked ape".

Should I really talk about how annoying the ads are? There are adblocks for internet browsers, we should have one for real life.

In big cities, people talk every day about this trait of the city - to tire you down. The unnatural number of people and the advertising posters anywhere you look play a large part in it. This leads you to put a lot of thought in behaving properly with every new person you see in addition to constant distraction from catchy advertisings of products you probably don’t need in your life.

Successful augmented reality glasses software will be able to hide advertisements and human faces, which will hopefully reduce stress on the inhabitants of the modern city.

# 1 Task description

The goal of this thesis project is to develop mathematical and software solutions for hiding faces and advertising banners on video in real time.

Functional requirements to the software:

1. The system should be able to read data from a webcam of a user.
2. The system should be able to detect faces.
3. The system should be able to detect advertising.
4. The detection results should be presented to the user graphically as a live video stream with hidden faces and ads.

The developed software running on modern PC with the Nvidia GeForce GTX 1650 GPU or better should meet the following non-functional requirements:

1. Face and advertising detection should be real time.
2. Face and advertising detection should run 24+ frames per second (counting separately).
3. The delay between the processed video and reality should not exceed 50 msec.

# 2 The analysis of existing methods and datasets

### 2.0.1 On non-machine learning based approaches

Past generation methods of object detection often rely on hand-crafted pixel maps. These maps are supposed to represent the distinctive features of the object we want to detect. Maps for the detection of objects like faces or advertising should get really sophisticated and numerous in order to do the job. But how sophisticated can we make them and can we come up with all the necessary ones for detection?

Today’s convolutional neural networks can find these distinctive features automatically. Not only this makes it is easier to program, but also gives greater accuracy. This makes non-ANN based methods look obsolete. Maybe, the only competition older methods can still give CNNs is in speed, but with even a modest GPU speed is neither a problem for CNNs.

### 2.0.2 Obtaining GPU results

In this chapter we will present fps results of performing inference with machine learning models on various GPUs. These results are mostly found in the original articles covering the method or on Github repositories. Some of the results were inferred from others by multiplying them on GPU coefficients. These coefficients are calculated using fps data from a GPU benchmark website [www.gpucheck.com](http://www.gpucheck.com).

GPU coefficients are calculated as

where fx is the fps metric of the GPU, is the fps of the Nvidia GeForce GTX 1650.

We use the following coefficients:

|  |  |
| --- | --- |
| GPU | C |
| Nvidia TESLA k80 | 0.45 |
| Nvidia 1050Ti | 0.69 |
| Nvidia Titan X | 1.62 |
| Nvidia 1080Ti | 2.15 |
| Nvidia GTX 1650 | 1 |

## 2.1 Datasets

### 2.1.1 Advertising dataset

Figure 2.1-2.2. Samples from ads dataset.

Our advertising dataset consists of Google Street View screenshot images. Annotation for each image is a list of pentagonal bounding boxes’ coordinates enclosing ads banners. This dataset is made for solving instance segmentation (pixelwise object detection). It is a more difficult variant of the task of object detection (enclosing object in rectangular box).

The dataset contains 6594 images, 4551 of which have no ads in them.

### 2.1.2 Faces dataset

Figure 2.3-2.4. Samples from WIDER FACE dataset.

WIDER FACE is a benchmark dataset for evaluating the accuracy of face detection models. Its images are selected from the WIDER dataset. The benchmark quality of the dataset emerges from the fact that owners of the WIDER FACE do not release the ground truth label values for the test set. Images are also split into 3 groups by difficulty: easy, medium and hard.

There are 32 203 images with 393 703 labeled faces with high degree of variability in scale, pose, and occlusion.

## 2.2 Advertising detection

### 2.2.1 Mask R-CNN

Mask R-CNN is a modification of another object detection architecture called Faster R-CNN. The title of the article to the Faster R-CNN reads “Towards Real-Time Object Detection with Region Proposal Networks”. But this model does not actually run in real time, that is on regular computers.

The biggest strength of this model is the fact that it is made to solve instance segmentation. This means it has great detection accuracy and it is just the right model for our ads dataset. But this feature comes at a cost of slow inference time and hence low fps.

Mask R-CNN performance:

|  |  |  |
| --- | --- | --- |
| GPU | fps | Accuracy |
| Nvidia TESLA k80 | 4 | 35.7 |
| Nvidia 1050 Ti | 7 |
| Nvidia GTX 1650 | ~9 |

### 2.2.2 YOLACT

YOLACT is a YOLO based neural network architecture for solving instance segmentation task. The authors of this model tried to keep a high speed of YOLO architecture while adding the segmentation capability.

The model has 35% lower accuracy then Mask R-CNN but, since it solves segmentation, the accuracy is still up there.

YOLACT performance:

|  |  |  |
| --- | --- | --- |
| GPU | fps | Accuracy |
| Nvidia TESLA k80 | ~14 | 26.8 |
| Nvidia 1080 Ti | 37 |
| Nvidia GTX 1650 | ~17.2 |

### 2.2.3 YOLO

We have already covered YOLACT, one of the successors of YOLO. Let’s now find out what YOLO is.

You Only Look Once is a breakthrough neural network architecture capable of real time object detection. Compared to its competitors at that time, At the time it was released, the competition was dominated by architectures like Fast R-CNN (region based CNN), which perform classification on image regions proposed by CNN. These models end up performing prediction multiple times for same regions in an image. YOLO architecture consists almost entirely of convolutional layers which makes it just a big CNN. It passes an image (nxn) once through the FCNN (fully convolutional neural network) and the output is (mxm) prediction. This architecture is splitting the input image in mxm grid and for each cell it generates 2 bounding boxes and class probabilities for those bounding boxes. Note that bounding box can easily be made larger than the grid cell itself.

YOLOv1 performance:

|  |  |  |
| --- | --- | --- |
| GPU | fps | mAP 0.5 |
| Nvidia Titan X | 45-150 | - |
| Nvidia GTX 1650 | 27-92 |

### 2.2.4 YOLOv3

YOLOv3 is a 3rd iteration of the state-of-the-art YOLO algorithm. There seems to be no point in considering YOLOv2 because YOLOv3 is evidently better. But let’s cover the main differences between YOLO and YOLOv3.

Improvements upon vanilla YOLO:

1. Batch Normalization layer added after each convolution block. It has made training smother and increased mAP score.
2. Anchor Boxes introduced. They make YOLO be able to assign multiple classes to a single grid cell.
3. High resolution classifier. An improvement to the training process. In the past the classifier was trained on 224x224 images was immediately used to help perform detection on 448x448 images. The authors added an intermediate step: fine tuning (training starting from already trained model) at the full 448x448 resolution on ImageNet dataset for 10 epochs. This gives the network time to adjust its ﬁlters to work better on higher resolution input. This addition gives a 4% increase in mAP.
4. Better backbone CNN.
5. Predicting confidence score for each bounding box.
6. Increased small objects detection capability. It is achived by using short cut connections. They allow us to get finer-grained information from earlier feature map.

However! Detection of medium ad larger sized objects worsend.

YOLOv3 performance:

|  |  |  |
| --- | --- | --- |
| GPU | Fps | mAP |
| Nvidia 1080 Ti | 30-60 | 51.5 – 57.9 |
| Nvidia Titan X | 23-46 |
| Nvidia GTX 1650 | 14-28 |

### 2.2.5 YOLOv5

YOLOv5 is a 5th generation of YOLO and it’s the first unofficial one. We should also mention that since YOLOv4 arguably the most important person among the YOLO creators Joseph Redmon had left the research. In the original YOLOv3 paper and later in Twitter he expressed his concern about YOLO models being used for military purposes. Object detection is a bottleneck of many developments in automatic weapons. Making highly autonomous weapons can mitigate losses for the owner of the weapon while dramatically increasing enemy losses. If found in the wrong hands it will make a lot of bad things happen. The ease it takes to train a good object detector nowadays makes it a viable tool for terrorist organizations around the globe.

YOLOv4 had slight improvements to the training process, as well does YOLOv5.

* Accuracy: YOLOv4 has tiny advantage over v5.
* Speed: accurate YOLOv5 is faster and additionally there is a YOLOv5 version that is 3 times faster.
* Speed of training: YOLOv5 is about 20 times faster to train then v4.

Improvements upon YOLOv3:

* Various image augmentation techniques used during training.

YOLOv5 performance:

|  |  |  |
| --- | --- | --- |
| GPU | fps | mAP 0.5 |
| Nvidia 1080 Ti | 35-135 | 43-54 |
| Nvidia GTX 1650 | 16-62 |

## 2.3 Face detection

### 2.3.1 MTCNN

MTCNN is one of the most accurate models specifically built to solve face detection. It performs detection in multiple stages. When given an image MTCNN first off produces many candidate bounding boxes. Then two stage bounding box regression is performed. Lastly, facial landmarks including 2 eyes, nose and 2 lip corners are placed on the remaining boxes. The authors claim the model to be pretty fast.

MTCNN performance:

|  |  |  |
| --- | --- | --- |
| GPU | fps | Accuracy |
| - | 15 | Very high |
| Nvidia Titan Black | 99 |
| Nvidia GTX 1650 | 61 |

### 2.3.2 YOLO

Looking at how intricate MTCNN is, it may seem that face detection is too difficult of a task for all purpose detection model but it is not. Evidently, YOLO can show great on this task. Also, if we were to use YOLO for advertising and for faces, we could have merged the datasets and create a unified model that is twice as fast as 2 separate YOLOs.

|  |  |  |
| --- | --- | --- |
| GPU | fps | Accuracy |
| Nvidia GTX 1650 | 16-62 | High |

### 2.3.3 Viola-Jones method

Viola-Jones is one of the best pre deep learning era method for face detection that is sometimes being used to this day. It was introduced in 2001. On the 700MHz Intel Pentium III it was able to run at 15 frames a second. The original paper says there is a lot of room for increasing fps, so will trust them. Despite good framerate the accuracy is on the lower end comparing it to modern models.

|  |  |  |
| --- | --- | --- |
| GPU | Fps | Accuracy |
| Nvidia GTX 1650 | 15+ | Low |

## 2.4 Comparing methods

### 2.4.1 For advertising

After all the research we get the following list methods for advertising detection:

|  |  |  |  |
| --- | --- | --- | --- |
| Model | fps | mAP | Segmentation |
| YOLACT | 17.2 | 26.8 | Yes |
| Mask R-CNN | 9 | 35.7 | Yes |
| YOLO | 27-92 | 52-63 (VOC dataset) | No |
| YOLOv3 | 14-28 | 51.5-57.9 | No |
| YOLOv5 | 16-62 | 43-54 | No |

From the table we get the following results:

Both methods that have that very desirable trait of doing instance segmentation will give fairly low fps which is unacceptable.

YOLOv3 has the greatest accuracy among all other models. It has decent 28 fps which is enough if we were to run only one model at a time.

YOLOv5 has great (top 2) accuracy. Also, it can show great speed which can give us the opportunity to run multiple different models at the same time.

### 2.4.2 For faces

After all the research we get the following list methods for face detection:

|  |  |  |
| --- | --- | --- |
| Model | Fps | Accuracy |
| MTCNN | 30 | Very-High |
| YOLO | 16-62 | High |
| Viola-Jones | 15+ | Low |

From the table we get the following results:

MTCNN has the highest accuracy and gives good fps for running a single model at a time.

If we choose YOLOv5+MTCNN for our project MTCNN’s 30 fps will give only a small room of

ms

for YOLOv5 to do inference 24 times. On our GPU YOLOv5 will be able to do only

frames,

so half of what is needed.

If we choose YOLO+YOLO for our project, we will have no trouble with speed and models will be easier to train. Only the face detection accuracy will not be as good.

## 2.5 Conclusions

Previous analysis has led to the following conclusions:

1. 2 suitable datasets were found: containing images and annotations of faces and of ads respectively.
2. It was decided to use learning based methods as they are more accurate and ideologically better than other, which also makes them easier to code.
3. A list of methods has been proposed for solving face and ads detection tasks.
4. These methods were analyzed in terms of their fps and accuracy on the Nvidia GeForce 1650 GPU.
5. MTCNN was chosen for face detection.
6. YOLOv5 was chosen for advertising detection.

# Methodology description

In this chapter we will briefly cover the basics of artificial neural networks and convolutional neural networks. We will go more in depth when talking about YOLO and MTCNN methods.

## 3.1 Neural network

Artificial neural network (ANN) is a system design inspired by biological neurons in the brain. It is a basic structure of many modern object detection methods. The key feature ANNs inherit from neural networks in living organisms is the ability to learn. Same as our brain’s neural systems learn from stimuli coming through them from various sensory organs, ANN adapts to fit the data we send through it.

These ANN systems are built in a way so after being introduced to large amounts of training examples, they can learn to solve a task they were built to learn to solve.

Let’s start by looking at one of the simpler ANNs - binary classifier. The task this system is trying to solve is given an input vector return the probabilities of this input belonging to the first and second class. There are two classes, input vector consists of 3 values. Here is how our ANN architecture would look like

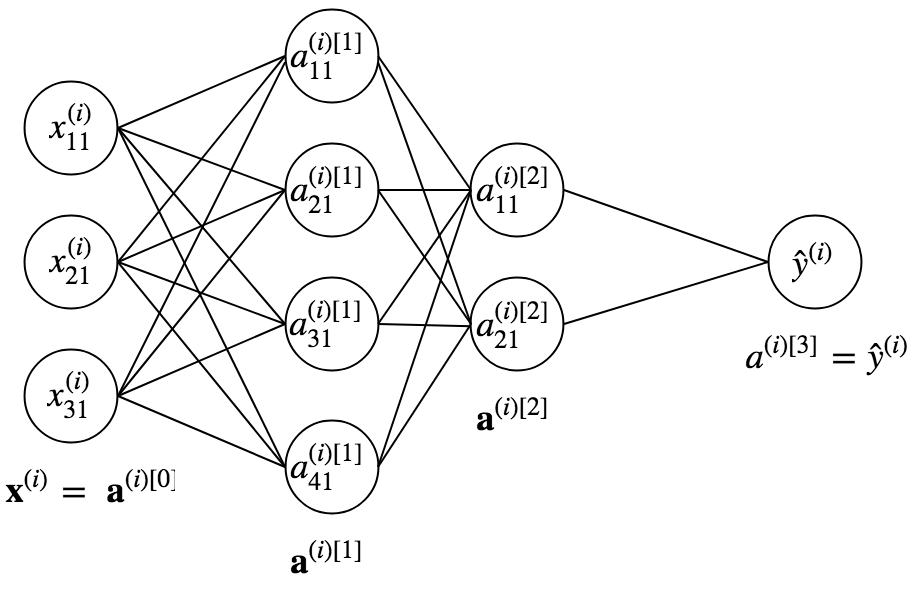


Figure 3.1. Classifier ANN architecture.

On the above graph (Figure 3.1) we can see neurons corresponding to our input value on the left side and the vector containing two values , [0,1]. Also, there are two intermediate layers on neurons, these are called hidden layers. But the essential parts of this model are really the connections between layers, these are called weights. Now let’s see how the output values appear from the input.

,

,

where l denotes the layer index, – training example, – the weight matrix, – the network input, – the output of the layer l, training sample . is the ReLU function applied element-wise to vectors. For the output layer the sigmoid activation function is used instead of ReLU, to get probabilities.

ReLU:

Sigmoid:

Forward propagation is calculating the output vector of the ANN from the input vector.

Using the above-mentioned formulas for forward propagation we get the output, but it is not going to be meaningful the first time we run it. In order to output better results, we need to train our model. For training the backpropagation method is used.

## 3.2 Training

During training the model is introduced to the training samples one by one. Each sample is fed as an input to the network and the loss (error) between the network prediction and the ground-truth label is calculated.

To find the loss (error) of the prediction we use binary cross-entropy

,

where y is the ground-truth, is a prediction from ANN.

Then we use backpropagation to find the gradient of the loss. After that we use gradient decent to correct the weights values using the found gradient.

,

where is learning rate.

## 3.3 Convolutional neural network

CNN is highly effective while working with image data. The classic task that can be solved by CNN is image classification. These network architectures are successful thanks to their ability to automatically find and remember defining features in training images, something that people tried to do manually in the past.

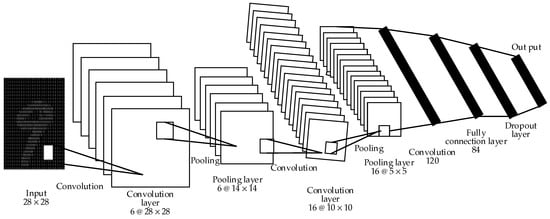


Figure 3.2. Example CNN pipeline architecture.

On the figure above (Figure 3.2) we can see that CNNs usually consist of convolutional, pulling, fully-connected and dropout layers. Let’s examine those in more detail.

### 3.3.1 Convolutional layer

The convolutional layer consists of kernels which are two dimensional arrays. The convolution is a process of creating a new array by sliding kernels across all the cells of the input array and for each patch computing

Note that kernel that generates the feature map is shared for all cells. This sharing mechanism has some major advantages, particularly it can reduce the number of parameters and make the network easier to train.

The activation value of a feature can be computed as

The activation functions used in CNNs are usually sigmoid, tanh or ReLU.

The loss function is the same as for non-convolutional networks

An example of a kernel is the following 3x3 array (Figure 3.3). It can be responsible for detecting crosses

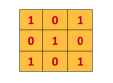


Figure 3.3. Example of kernel.

Convolutional layers have these parameters:

*Size* – length of the side of a square filter.

*Padding* – pixel thickness of the frame padding. Padding is added to the input arrays in order to improve feature facilitation closer to the edge.

*Stride* – the number of pixels the kernel shifts every time.

### 3.3.2 Pooling layer

Pooling layer is used to extract the most valuable parts of the input signal array. It is often used to decrease the dimensions of the convolutional layer output, decreasing in turn the overall number of model parameters. Useful if the input image is large.

Pooling layer works similarly to convolutional layer in that the mask of aggregation comes across the array; on each stop the largest value under the mask is saved in the output array area corresponding to the mask position. For example, having a mask of a size and a stride (step) 2 the output of this pooling layer will be 4 times smaller than the input. Example of pulling layer (Figure 3.4)

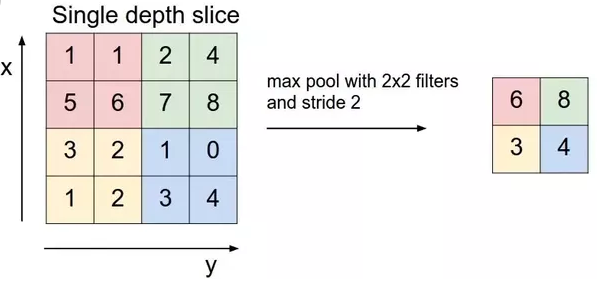


Figure 3.4. Pooling layer in action

Pooling layers have the majority of parameters convolutional layers have.

### 3.3.3 Dropout layer

Dropout layers “turn off” a certain percentage of the incoming connections. This makes the model adjust its parameters so it doesn’t depend on just a small part of the network.

Using dropout layers decreases model’s chances of overfitting, so it’s a regularization technique. It is put after the fully-connected layer. Because FC layers have a lot of parameters, they tend to just remember the training samples losing the ability to generalize on test data.

## 3.4 Intersection Over Union

Intersection over Union, or , is a function that quantifies the overlap between two bounding boxes. It is used in object detection to check how correctly a predicted bounding box ​ is positioned over the ground-truth bounding box ​. It is defined as:

Remark: IoU[0,1]. Often, by convention, predicted bounding box Bp​ is considered being reasonably good if ≥0.5.

Examples of calculating IoU (Figure 3.5)



Figure 3.5. IoU examples.

## 3.5 Non-max suppression

Non-max suppression is a method used to reduce the number of bounding box detections for a single object to a minimum.

Each output prediction is , where is confidence.

The algorithm in pseudocode:

Discard all bounding boxes with

While there are any detections left:

Take the bounding box with the biggest . Output it as a prediction.

Discard all remaining boxes with IOU > 0.5 with the box from the previous step.

## 3.6 YOLO

### 3.6.1 CNN architecture

Architecture-wise YOLO algorithm is simply a big CNN with a lot of additional functionality (Figure 3.6).

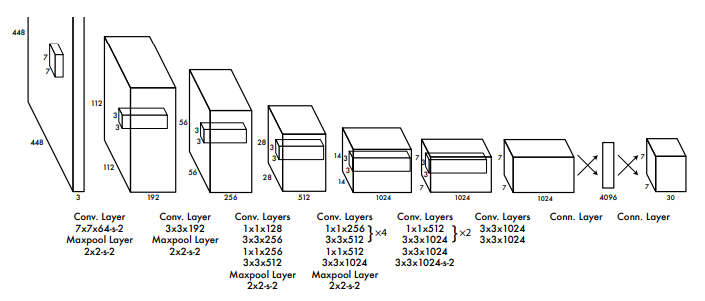


Figure 3.6. YOLO architecture.

Here is what the authors say about the CNN architecture of YOLO:

“Our detection network has 24 convolutional layers followed by 2 fully connected layers. Alternating 1 × 1 convolutional layers reduce the features space from preceding layers. For the final layer we use linear activation function and for all other layers the following leaky rectified linear activation is used:

”

### 3.6.2 Algorithm

Step 1: Divide the input image into grid.

Step 2: For each grid cell run the CNN that predicts the output vector .

where is the probability of detecting an object, coordinates of the center of a prediction, are its dimensions, and are class detections and k is the number of anchor boxes.

Step 3: Apply non-max suppression algorithm to discard any potentially overlapping bounding boxes.

Note: if then corresponding should be ignored.

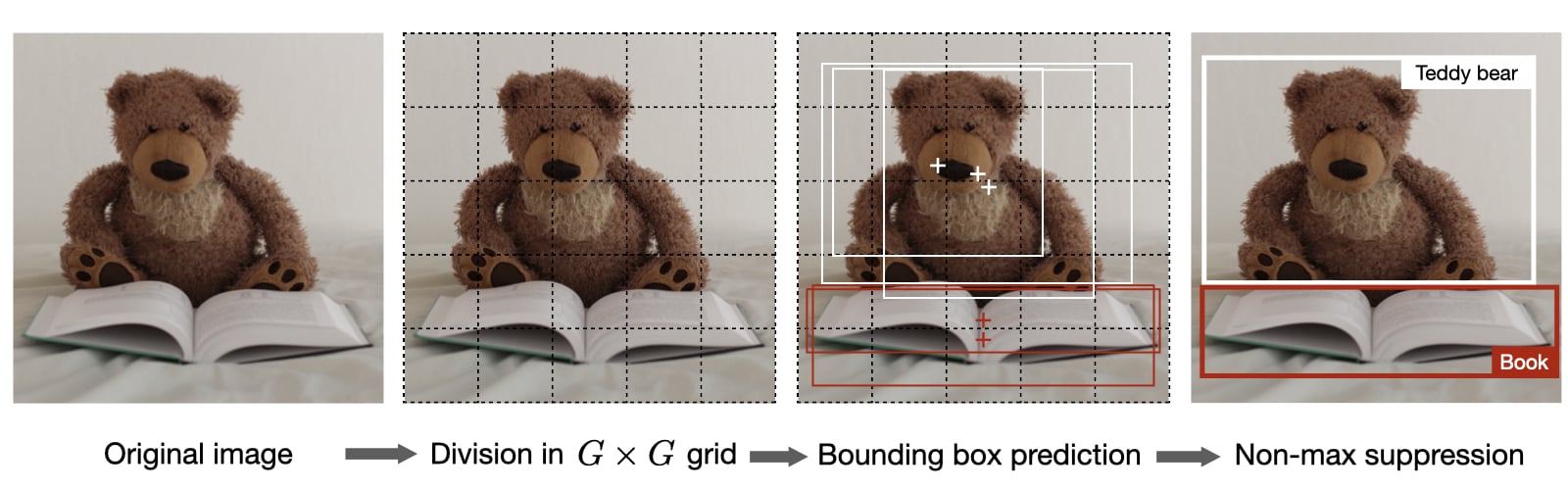


Figure 3.7. YOLO algorithm work example.

### 3.6.3 Anchor boxes

Anchor boxing is a method used to predict overlapping bounding boxes. Old object detection methods used to be able to predict only one object per cell/image. Newer versions of YOLO can predict more than one. To do this they set a new output layer, where confidences, coordinates, dimensions and classes for every box are contained in a single vector

Old: New:

Additionally, each positional box in the output vector corresponds to the box from a set of predefined anchor boxes. For instance, this set can contain two boxes: one is more horizontal and the other more vertical, so that they predict different objects more accurately (Figure).

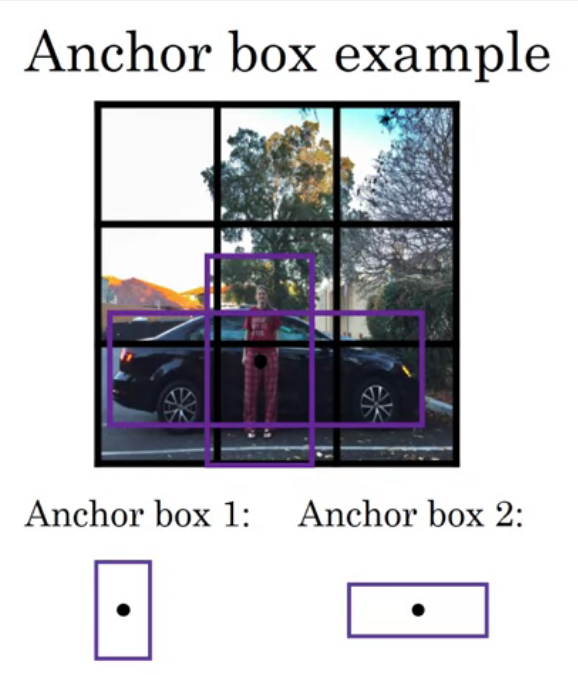
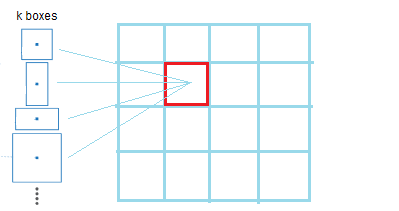
 

Figure 3.8. Anchor box examples

### 3.6.4 Training

Loss function:

where denotes if object appears in cell and denotes that the th box in cell is “responsible” for that prediction.

Let’s look at this function line by line:

The first term is the residual sum of squares error between predicted () and ground truth () coordinates. The sum is over all grid cells of an image and all the boxes corresponding to this cell.

The second term is similar to the first term, but instead of error in box center coordinates we calculate the error in width and height in the box dimensions. Square root is used to reﬂect that small deviation in large boxes matter less than in small boxes.

Confidence error, where and 0≤C≤1

if there is no object in the grid cell or the box is not responsible for detecting that object.

This term accounts for pushing the confidence score for cells that don’t contain objects to zero. Because often many grid cells do not contain objects the training can diverge early. To remedy this the parameter is used to decrease the loss. Positional and dimensional variables are not concerned because there is no object.

The fifth term sums the errors in probabilities for all the classes for all the grid cells .

## 3.7 Measuring accuracy

To measure the accuracy of object detection researchers also use the AP (Avarage Precision), AR (Avarage Recall) and mAP (Mean Average Precision) metrics.

Precision is defined as

,

where respectively is the number of true positive and false positive detections (Figure).

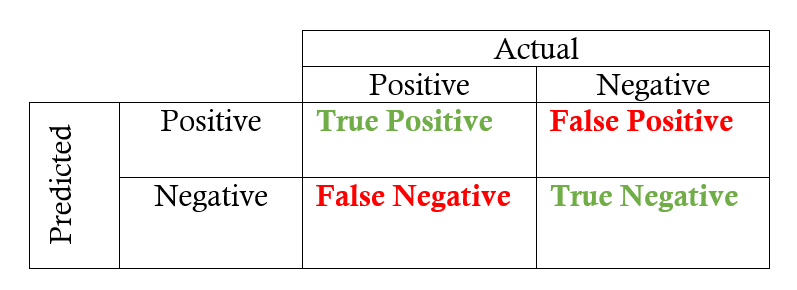


Figure 3.9. Kinds of detections.

So precision is the fraction of accurately made predictions with respect to all predictions. This metric does not account for the predictions that were not made. A model that fails to detect any object can have high precision.

Recall is defined as

where respectively is the number of true positive and false negative detections (Figure).

Recall is the fraction of accurately detected objects with respect to all positive cases in reality.

Average Precision (AP) is defined as

where r is a recall value and is the precision value corresponding to . Multiple precision-recall pairs appear when changing the threshold for a score to signal positive detection.

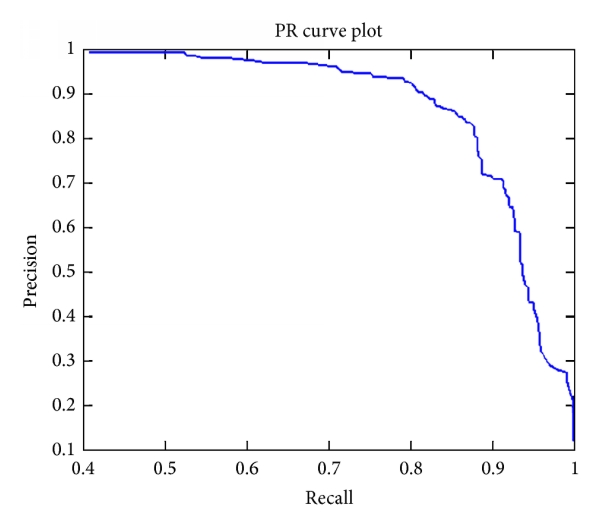


Figure 3.10. Precision-recall curve.

Mean average precision (mAP) is the average value for all AP values across all classes.

Sometimes, researchers put numbers after AP like AP50, AP70, etc. These numbers denote the threshold IOU value used to separate the false positive from the true positive.

## 3.8 Data augmentation

Dataset augmentation is a technique the goal of which is to increase performance of the model by altering samples in the dataset.

Here we will look at some of the existing augmentation methods.

### 3.8.1 Distortion

Photometric Distortion — This includes changing the brightness, contrast, saturation in an image or adding noise to it.

Geometric distortion – This includes random flipping, scaling, rotating. When using these methods, we must precisely update bounding boxes.

### 3.8.2 Image occlusion

Random erase – This includes cropping random rectangular parts of the image. In place of the crops the noise or the mean dataset pixel value is put. Usually, it is implemented with varying proportions of the cropped area. This technique prevents the model to memorize the training data and is used to reduce overfitting.



Figure 3.11. Examples of random erase [17]

Hide and seek – Image is divided into grid SxS cells. Then each cell is made hidden with certain probability (p\_hide). This makes the model learn to detect the object by its different parts, not just one. Technique is used to reduce overfitting.

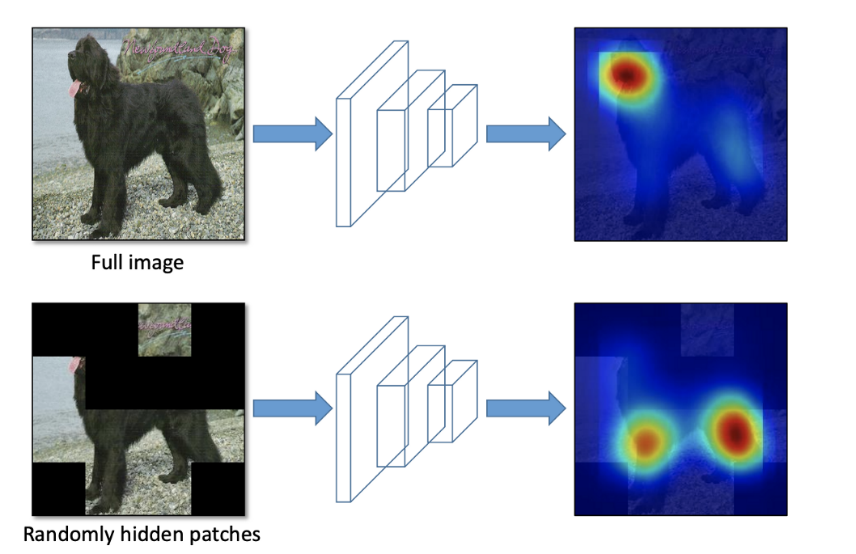


Figure 3.12. The impact of using hide and seek in training CNN.[18]

Grid mask – Blocks an image with a sparse grid of squares. Similar to hide and seek, this makes the model learn to account for the component parts of the image.

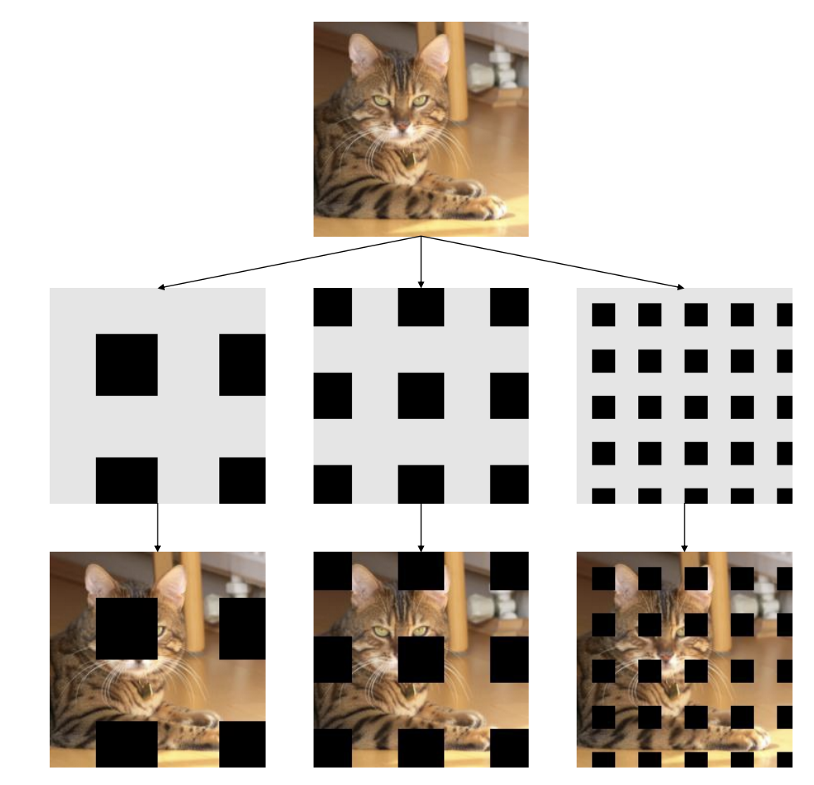


Figure 3.13. Grid masking example.[21]

MixUp – Overlaying two images and their labels making them both visible by equally reducing their opacities.

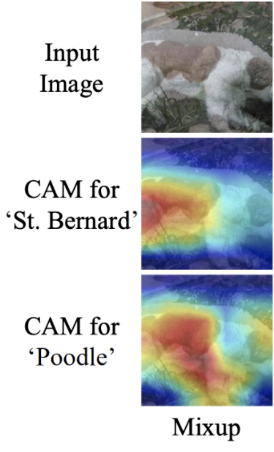


Figure 3.14. Impact of using MixUp [20]

### 3.8.3 Augmentations for YOLO

CutMix – Combining images by pasting parts of one image on the other. Works well if we need to make model to recognize not just one part of the image, e. g. a head of a dog. It is similar to Erase method but the erased part is substituted with another image along with its ground-truth labels. The ratio of each image is set in the generation process. It was found that this method is more effective than just Erase and MixUp methods.

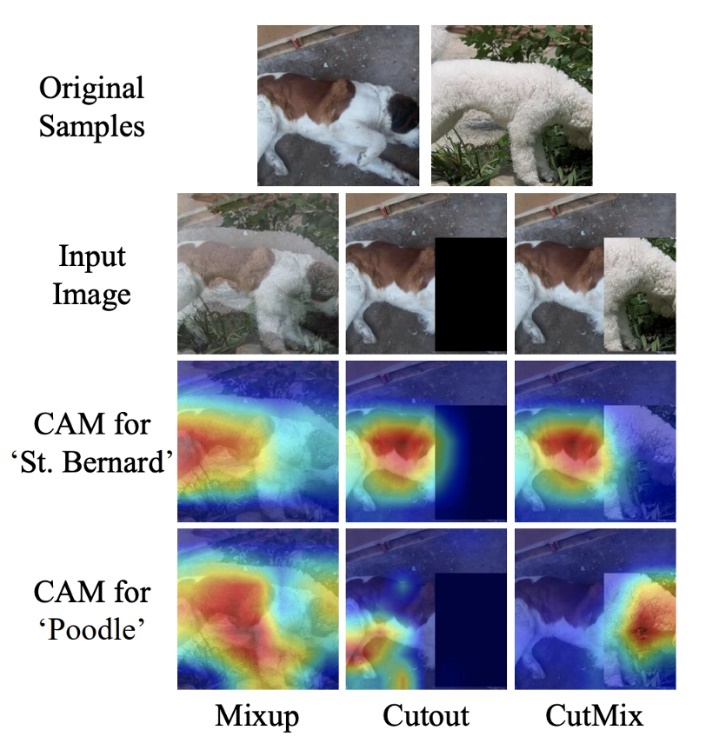


Figure 3.15. Impact of CutMix example.[19]

Mosaic data augmentation – Combines 4 training images into one in certain ratios. This technique can help with the detection of smaller scale objects in an image as well as localizing different types of images I different parts of the frame.

Class label smoothing – Tweaks the ground-truth probabilities to be less strict. This is done with an intuition that when a model becomes overly sure with a prediction close to 1.0, it is often wrong, overfit, and over looking the complexities of other predictions in some way. Authors of YOLO are using 0.9 probability instead of 1.

Self-Adverserial Training (SAT) This is a technique that uses the data about the state of the model to construct a training example the model will struggle the most with. The new transformed image is created using the loss signal.

## 3.9 Face detection

Face detection is an especially difficult detection task due to high variety in features e. g. different shadings on a face, glasses, facial hair, other accessory attributes, ethnicity of a person and angle of view, facial expression.

Viola-Jones algorithm was a breakthrough way of solving face detection having both real time speed and good accuracy for the time. But Viola-Jones suffers from just little face tipping or turning, different lighting conditions and ethnicities. This algorithm relies on hand crafted feature maps so it is bound to be bested by deep CNN approaches. CNN methods neglect all these variations in features given enough training samples. Besides, we can go further than just CNN detection. In recent years, researchers were trying to expand CNNs approach with pattern recognition and the incorporation of inherent correlations between facial features. Multi task cascaded convolutional networks (MTCNN) architecture is the result of that.

* Face alignment is a computer vision task the goal of which is to find the coordinates of a given landmarks points on the face e. g. eyes, tip of the nose, corners of the mouth etc.

## 3.10 MTCNN

Multi task cascaded convolutional networks (MTCNN) is a deep convolutional neural network cascaded architecture made to solve face detection. It exploits the inherent correlation between face detection and face alignment to increase the accuracy of both tasks. This model consists of three staged CNN (Figure) that predicts face and landmark location in a coarse-to-fine manner.

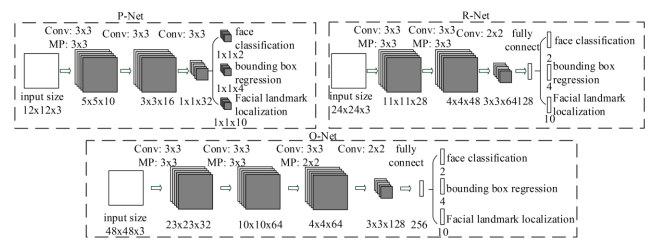


Figure 3.16. Architecture of MTCNN. [5]

### 3.10.1 Algorithm

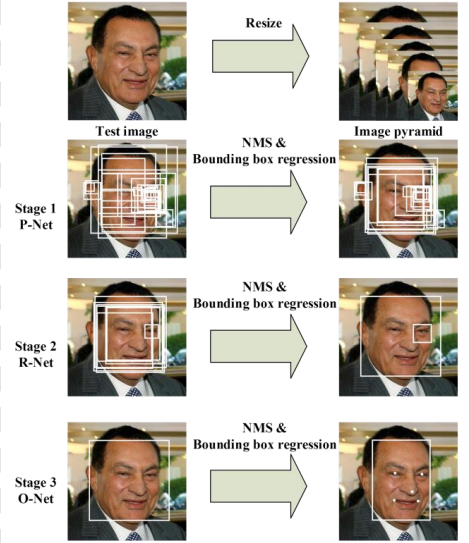


Figure 3.17. Pipeline of the MTCNN that includes three-stage multi-task deep convolutional networks. Firstly, candidate windows are produced through a fast Proposal Network (P-Net). After that, they are refined in the next stage through a Refinement Network (R-Net). In the third stage, The Output Network (O-Net) produces final bounding box and facial landmarks position. [5]

Firstly, after getting an image it is being resized to different scales creating a pyramid of images.

The pyramid then becomes an input of the 3-staged cascaded CNN:

Stage 1: “Fully convolutional network” called Propositional network (P-Net) is used for getting candidate bounding boxes, specifically their coordinates and so called “regression vectors”. Then the estimated bounding box regression vectors are used to calibrate the candidates. Then non max suppression is used to discard overlapping boxes.

Stage 2: All candidate bounding boxes are fed into Refine-Network (R-Net). It further rejects a number of candidates performing the same regression and non max suppression steps.

Stage 3: This stage is similar to stage 2, but additionally it tries to find and align 5 facial landmarks’ position within each bounding box.

### 3.10.2 Training

There are cumulatively three types of tasks pointed out to train the CNN detectors for on each of the stages

1. Face classification: The learning objective is formulated as a two-class classification problem. For each sample we use cross-entropy loss:

,

where is the probability produced by the network indicating sample being a face; in {0,1} denotes the ground-truth label.

1. Bounding box regression: For each candidate window the model should predict the offset between it and the nearest ground truth. The learning objective formulated as a regression problem. We use Euclidean loss for each sample :

*,*

where is regression target obtained from the network and is ground-truth coordinate. There are 4 coordinates thus .

1. Facial landmark localization: Similar to the bounding box regression we formulate a regression problem with Euclidean loss:

,

where is a facial landmark coordinates received from the network and is the ground-truth coordinate. Five facial landmarks are used, thus .

1. Multi-source training: Since CNNs on every particular stage works with different inputs e. g. face image, non-face image, image partially aligned on a face. In this case, some of the loss functions (1-3) are not used. Then the overall learning function can be formulated as:

where N is the number of training samples. corresponds to the task importance. The constants set is used P-Net and R-Net, while used in O-Net for more precise facial landmark localization. is the indicator of type of the sample. This method is using stochastic gradient descent to train the CNN.

## 3.11 Conclusion

This section briefly defines what an artificial neural network is, its characteristics on the example of a binary classifier. The principle of operation of the convolutional neural network was described. After that, the concept of object detection is introduced. It has been described how image classification methods have inspired methods for detecting objects in images.

Moving on to object detection methods, IoU, AP, RP, mAP detection quality metrics were defined. An auxiliary method for detection was considered: non-maximal suppression. The YOLO and MTCNN methods were described in detail. Dataset augmentation methods were also described in detail.

# 4 Developed software

## 4.1 Structural model

The software architecture was divided into 2 major parts: training the model (Figure 4.1) and using trained model (Figure 4.2).

### 4.1.1 Training

For training the model the image dataset needs to be prepared, that is

1. make a function to turn our instance segmentation dataset into detection dataset;
2. add augmented images to enhance the dataset;
3. split the data into training and validation parts and creating “train.txt” and “valid.txt” containing respective image paths.
4. resize it to match the CNN’s input shape.

Parts of YOLO architecture

To make the YOLO model it needs to be made part by part, that is

1. make function IOU;
2. make anchors module;
3. make non max suppression function;
4. get pre-trained CNN backbone;
5. make YOLO feedforward method;
6. method to do loss function and metric calculation;
7. method to update weights.

Similarly, to make MTCNN model the following steps should be made:

1. get pre-trained P-Net CNN;
2. get pre-trained R-Net CNN;
3. get pre-trained O-Net CNN;
4. assemble MTCNN.

Training the YOLO model can take a long time, like a couple of days, that is on a good GPU. To track the effectiveness of the training

1. save loss values scores;
2. plot the results using Tensorboard extension for Python.

To analyze the accuracy of the trained model and decide whether the accuracy is enough

1. save precision, recall and mAP scores;
2. plot the results using Tensorboard extension for Python.

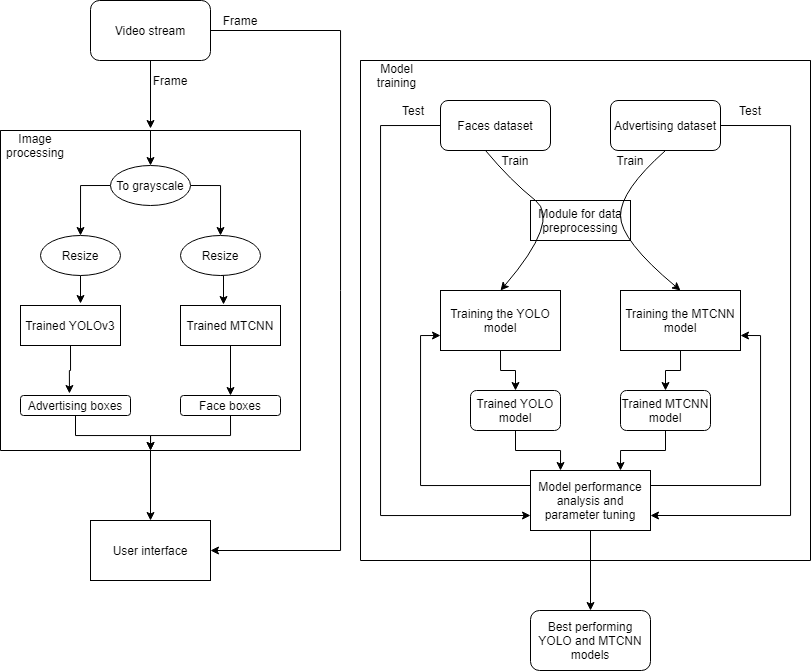


Figure 4.1. Model training diagram

### 4.1.2 Wrapping model into application

Now, after our models are trained, let’s use them to make a desktop application for a webcam which detects and blurs faces and advertising. For this

1. load models function to initialize the NNs once before doing inference;
2. a “detect” function that takes a frame image and returns bounding boxes for faces;
3. a function is needed to draw blurred bounding boxes on images;
4. create a window user interface with buttons to stop and run detection on camera input, perform

inference on a picture by path and record from webcam;

1. setup the data flow and detection from the webcam;
2. make an output window with real time stream.

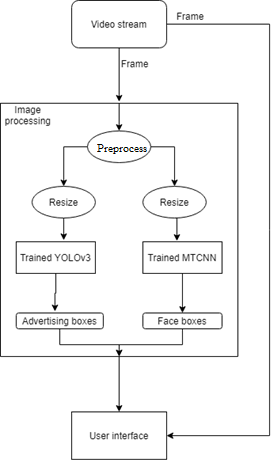


Figure 4.2. Model evaluation diagram

## 4.2 Dependencies

The software was written in Python language. the following libraries have been used:

* PyTorch, Numpy, SciPy for ANN architectures and GPU acceleration;
* OpenCV, Scikit-image, Pillow for image processing;
* Matplotlib, Tensorboard for data visualizations;
* PyQT for graphical user interface;
* Cython for speeding up Python code.

Also, Nvidia CUDA 10.1 used for GPU acceleration.

## 4.3 Evaluation

### 4.3.1 Training YOLO

When the script for training is being ran the following progress reports show up

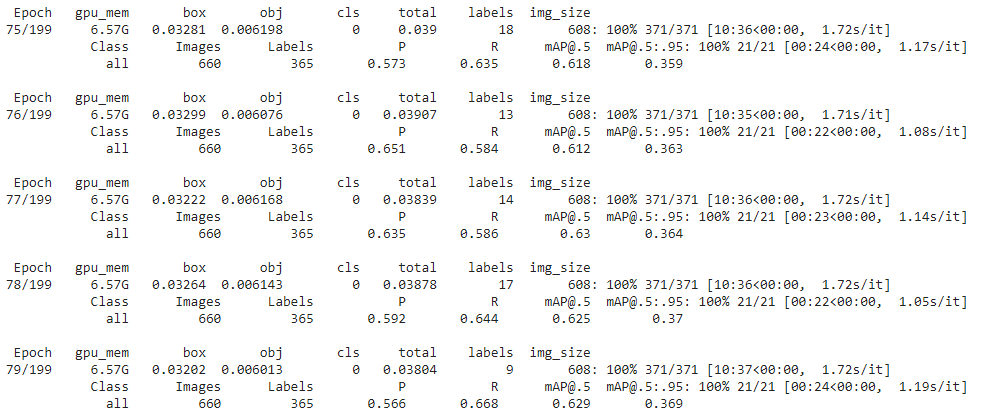
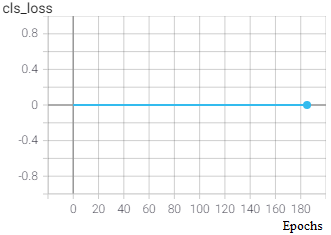
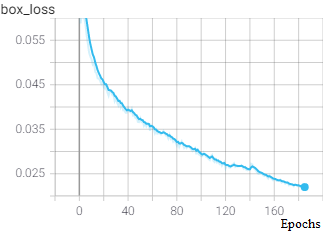
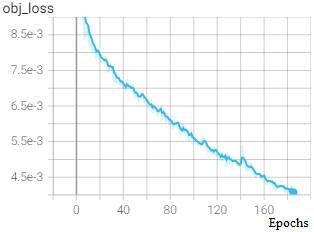


Figure 4.3. Training progress

Also, during the training there is an option to view the loss graphs in the Tensorboard extenstion



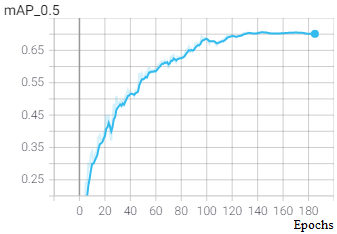
  
Figure 4.4. Loss after training for 185 epochs

From looking are looking at the 185 epochs loss graph of training, one can make out a downtrend on the box loss and object loss. Class loss remains constant, since there is only one class (advertising) in our training set and there is no “empty” class.

From the downtrend of the loss graph, it is concluded that the model’s accuracy increases on the training dataset, which is a sign of a healthy training. Now, if the loss becomes too low the model will overfit on the training data and won’t be able to generalize on test data. To make sure this is not happening here let’s move on to the validation.

### 4.3.2 Validation

During training the accuracy of the model has been tested on the validation dataset. Here are the results



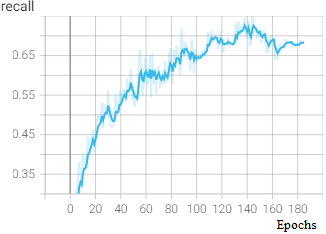
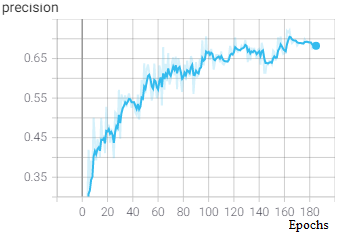


Figure 4.5. Training scores: mAP, precision and recall.

While looking at the 185 epochs graphs of precision, recall and mAP scores, one can see a stable increase in all scores. The mAP score plateaus at 70% and doesn’t decrease. This indicates the model has reached its peak performance and did not overfit.

So, it was managed to train an advertising detector that has 70% mAP accuracy. It was concluded that the score is pretty good and enough for the project’s needs.

## 4.4 Experimental results

When the application is ran, the following GUI shows up

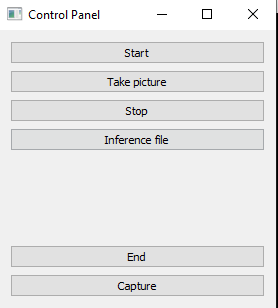


Figure 4.6. Application GUI

The GUI has the following buttons

1. “Start” to open webcam stream with blurred faces and ads;
2. “Take picture” to save a frame of the stream;
3. “Stop” to close the stream;
4. “Inference file” to choose file for face-ad detection and in app demonstration;
5. “End” to stop recording frame stream;
6. “Capture” to start recording frame stream.

In the main menu, if we press “start” we will see similar window stream

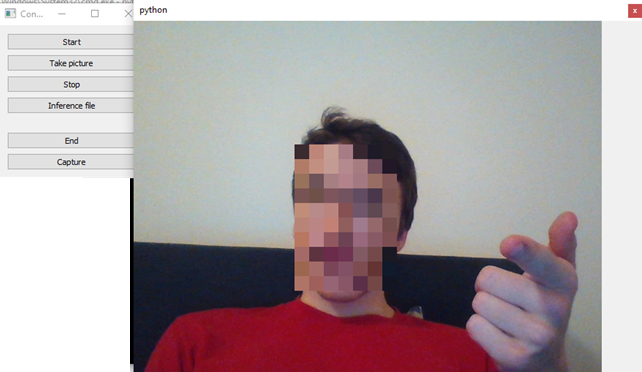


Figure 4.7. Processed video stream example

While looking at the stream we can see that our face gets detected and is being hidden.

To test the face detection NN we filmed a video of the program working, here are some of the frames

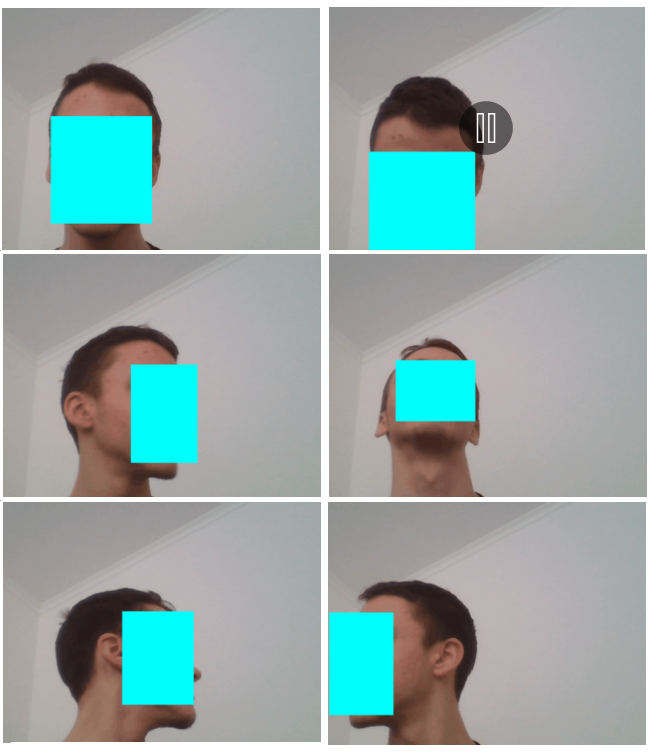


Figure 4.8. Frames with marked detections of faces from a webcam in real time.

While running it on our CPU we got detection speed of 2.2 detections per second. But when we run it on GPU Nvidia 1650 we get ~25 detections per second. This satisfies our requirements about real time performance. It is possible to make the stream even faster by also sometimes outputting frames without fresh detections.

Now let’s take a look at the ads detection frames of the same stream.

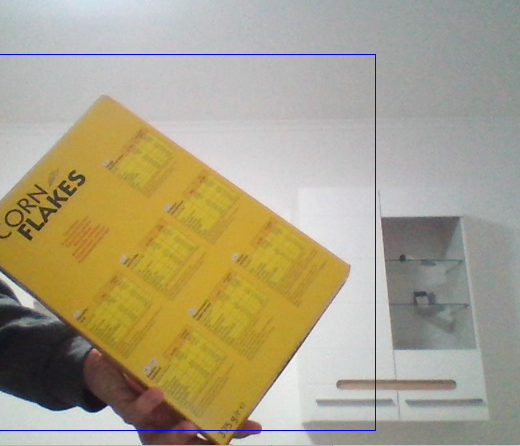


Figure 4.9. Frames with detections of advertising from a webcam marked with a blue rectangle.

The detection regions were not blurred so you can see what kind of ads it is able to detect. So, we’re looking at a fine rectangular detection of a corn flakes box, a kwas bottle label and a socks label. The program is working as expected without any major issues.

The following is a demonstration of showing faces and advertising to the program



Figure 4.10. Resulting frames with blurred ads and faces

Overall, our system definitely can detect faces and advertising. Ofcourse, not every advertising is detectable, because they can be pretty different. In fact, in our test we focused on detecting labels on food products and appliances, but the model was trained on completely different dataset of large street wall banners. Nevertheless, our labels were detected pretty accurately, although not without any effort, to be clear.

On the pictures above (Figure 4.10) we can see a face along with a phone label and a bread packet are being consistently detected. At the same, time we were not able to get to detect the salt packet at all times.

# 5 Future work

To improve the fps on weaker GPUs or, just in general, make the blurring smoother we can try using trackers. Tracking algorithms are designed in the following way. They are given an image and a bounding box for an object in this image so it remembers features of the object. Then, when it’s presented with the following frames it is able to quickly locate remembered object. The whole process should be much faster than inference time of an object detector. Also, we can perform tracking instead of detection for a greater speed up.

SiamFC++ architecture seems fit for this task.

# Сonclusions

The functional and non-functional requirements for our end product have been formulated. Summarizing, the program should run at 24+ fps on Nvidia GeForce 1650 GPU, it should be able to accurately detect faces and advertising in incoming images and it should have a GUI.

The advertising dataset made for instance segmentation has been found and turned to a detection dataset.

A list of object detection methods has been analyzed. The methods were compared in terms of speed and accuracy. It was chosen to use MTCNN model for face detection and YOLOv5 model for advertising detection. Note: There was another optimal variant that is to use a single unified YOLO model for both face and advertising detection. This one would give much higher fps and training speed while suffering a little more with face detection accuracy.

The YOLO model has been trained using the ads dataset. The loss value was being analyzed during the training. The training was stopped when the saturation point was reached. The trained model was evaluated on the validation dataset. The YOLO model has shown 70% mAP50 accuracy which was decided being enough.

A desktop user interface to use our models was developed. The program was developed for streaming processed webcam video in real-time, 24 fps.

# References

1. Redmon, J., Divvala, S., Girshick, R., & Farhadi, A. You Only Look Once: Unified, Real-Time Object Detection. *IEEE Conference on Computer Vision and Pattern Recognition (CVPR).* doi:10.1109/cvpr.2016.91, 2016.
2. Redmon, J., & Farhadi, A. YOLO9000: Better, Faster, Stronger. IEEE Conference on Computer Vision and Pattern Recognition (CVPR). doi:10.1109/cvpr.2017.690 , 2016.
3. YOLOv3: An Incremental Improvement. arXiv:1804.02767, 2018.
4. Chen, W., Huang, H., Peng, S., Zhou, C., & Zhang, C. YOLO-face: a real-time face detector. The Visual Computer. *doi:10.1007/s00371-020-01831-7*, 2020.
5. Zhang, K., Zhang, Z., Li, Z., & Qiao, Y. Joint Face Detection and Alignment using Multi-task Cascaded Convolutional Networks. In [IEEE Signal Processing Letters](https://ieeexplore.ieee.org/xpl/RecentIssue.jsp?punumber=97)*,* 23(10), 1499–1503 *doi:* [*10.1109/LSP.2016.2603342*](https://doi.org/10.1109/LSP.2016.2603342)*,* 2016.
6. Yang, S., Luo, P., Loy, C. C., & Tang, X. WIDER FACE: A Face Detection Benchmark. IEEE Conference on Computer Vision and Pattern Recognition (CVPR).  WIDER FACE: A Face Detection Benchmark. doi:10.1109/cvpr.2016.596, 2016.
7. YOLACT: Real-time Instance Segmentation. *arXiv:1904.02689*, 2019.
8. Mask R-CNN. *arXiv:1703.06870*, 2018.
9. Yinda Xu, Zeyu Wang, Zuoxin Li, Ye Yuan, Gang Yu. SiamFC++: Towards Robust and Accurate Visual Tracking with Target Estimation Guidelines. Conference on Artificial Intelligence (AAAI-20), 10.1609/aaai.v34i07.6944, 2020
10. Erik-Linder Noren, PyTorch YOLOv3, (2018), Github repository.

https://github.com/eriklindernoren/PyTorch-YOLOv3

1. Tim Esler, Face Recognition using PyTorch, (2020), Github repository.

<https://github.com/timesler/facenet-pytorch>

1. Advertising dataset

[Electronic resource] / Amir R Zamir, Mubarak Shah. 2012. Access mode:

<http://www.cs.ucf.edu/~aroshan/index_files/Dataset_PitOrlManh/zipped%20images/part1.zip>

1. Annotations for advertising:

WannaFly, Mask\_AD, Github repository.

<https://github.com/WannaFIy/mask_AD/releases/download/AD/regions.csv>

1. Online course “Convolutional Neural Networks”

[Electronic resource] / deeplearning.ai; Coursera. 2017. Access mode: https://www.coursera.org/learn/convolutional-neural-networks

1. “Convolutional neural networks cheatsheet”

[Electronic resource] / [Afshine Amidi](https://twitter.com/afshinea) and [Shervine Amidi](https://twitter.com/shervinea); Stanford. 2018. Access mode: <https://stanford.edu/~shervine/teaching/cs-230/cheatsheet-convolutional-neural-networks>

# Appendix А. Code listing.

The complete structured version of the code with user guide can be found on our Github: https://github.com/deepwebhoax/face\_ad\_blur

qtapp.py GUI

import sys

from PyQt5 import QtGui, QtCore

from PyQt5.QtWidgets import QWidget, QMainWindow, QApplication, QPushButton, QLabel, QVBoxLayout, QFileDialog

import cv2

import threading

from PIL import Image

import time

import os, sys

sys.path.append(os.path.join(os.path.dirname(\_\_file\_\_), 'yolov5'))

from helpers import mosaic\_blur\_multiple

# from mtcnn\_model import extract\_faces, draw\_rectangles

from yolov5 import detect

from facenet\_pytorch import MTCNN

from mtcnn\_detect import extract\_faces

model\_face = MTCNN(keep\_all=True, post\_process=True).eval()

def to\_bytes(non\_bytes\_img, type='jpeg'):

    from io import BytesIO

    buf = BytesIO()

    non\_bytes\_img.save(non\_bytes\_img, type)

    buf.seek(0)

    image\_bytes = buf.read()

    buf.close()

    return image\_bytes

def swapxy(faces, shape, toint=True):

    for i, f in enumerate(faces):

        # faces[i] = [max(f[1], 0), max(f[0], 0), min(f[3], shape[0]), min(f[2], shape[1])]

        faces[i] = [max(f[0], 0), max(f[1], 0), min(f[2], shape[0]), min(f[3], shape[1])]

        if toint:

            faces[i] = [int(o) for o in faces[i]]

    return faces

class QtCapture(QWidget):

    def \_\_init\_\_(self, \*args):

        super(QWidget, self).\_\_init\_\_()

        self.fps = 24

        self.cap = cv2.VideoCapture(\*args)

        self.video\_frame = QLabel()

        lay = QVBoxLayout()

        lay.setContentsMargins(0,0,0,0)

        lay.addWidget(self.video\_frame)

        self.setLayout(lay)

        self.isCapturing = False

        self.ith\_frame = 1

    def setFPS(self, fps):

        self.fps = fps

    def nextFrameSlot(self):

        ret, frame = self.cap.read()

        # Save images if isCapturing

        if self.isCapturing:

            cv2.imwrite('img\_%05d.jpg'%self.ith\_frame, frame)

            self.ith\_frame += 1

        frame = cv2.cvtColor(frame, cv2.COLOR\_BGR2RGB)

        # Face detection

        face\_detections = model\_face.detect(frame)

        if face\_detections[0] is None:

            fixed\_detections\_xyxy = []

        else:

            fixed\_detections\_xyxy = swapxy(list(face\_detections[0]), shape=frame.shape[:2])

        # face\_detections = extract\_faces(frame)

        # Advertising detection

        # ad\_detections = model\_ad.detect(frame)

        ad\_detections = detect.detect\_custom(frame, weights='yolov5/weights/best.pt').tolist()

        trimed\_ad\_detections = [det[:4] for det in ad\_detections]

        # ad\_detections = []

        # print('ad', ad\_detections)

        # print('face', fixed\_detections\_xyxy+trimed\_ad\_detections)

        detections = fixed\_detections\_xyxy + trimed\_ad\_detections

        print(detections)

        self.blur = True

        # Draw detections on image

        if self.blur:

            frame = mosaic\_blur\_multiple(frame, detections)

        else:

            print(detections)

            for x1, y1, x2, y2 in detections:

                cv2.rectangle(frame, (int(x1), int(y1)), (int(x2), int(y2)), color=(0,0,255))

        # t1 = threading.Thread(target=QtGui.QImage, args=(frame, frame.shape[1], frame.shape[0], QtGui.QImage.Format\_RGB888))

        # t1.start()

        img = QtGui.QImage(frame, frame.shape[1], frame.shape[0], QtGui.QImage.Format\_RGB888)

        pix = QtGui.QPixmap.fromImage(img)

        self.video\_frame.setPixmap(pix)

        self.count += 1

        print((time.time()-self.t1)/self.count)

    # def streamThread(self, frame, frame.shape[1], frame.shape[0], QtGui.QImage.Format\_RGB888):

    #     while

    def start(self):

        self.t1 = time.time()

        self.count = 0

        self.timer = QtCore.QTimer()

        self.timer.timeout.connect(self.nextFrameSlot)

        self.timer.start(1000./self.fps)

    def stop(self):

        self.timer.stop()

    # ------ Modification ------ #

    def capture(self):

        if not self.isCapturing:

            self.isCapturing = True

        else:

            self.isCapturing = False

    # ------ Modification ------ #

    def deleteLater(self):

        self.cap.release()

        super(QWidget, self).deleteLater()

class ControlWindow(QWidget):

    def \_\_init\_\_(self):

        QWidget.\_\_init\_\_(self)

        self.capture = None

        self.start\_button = QPushButton('Start')

        self.start\_button.clicked.connect(self.startCapture)

        self.picture\_button = QPushButton('Take picture')

        self.picture\_button.clicked.connect(self.picture)

        self.quit\_button = QPushButton('End')

        self.quit\_button.clicked.connect(self.endCapture)

        self.end\_button = QPushButton('Stop')

        self.inference\_file\_button = QPushButton('Inference file')

        self.inference\_file\_button.clicked.connect(self.inf\_file)

        # self.dialog = QFileDialog()

        self.le = QLabel('')

        # ------ Modification ------ #

        self.capture\_button = QPushButton('Capture')

        self.capture\_button.clicked.connect(self.saveCapture)

        # ------ Modification ------ #

        vbox = QVBoxLayout(self)

        vbox.addWidget(self.start\_button)

        vbox.addWidget(self.picture\_button)

        vbox.addWidget(self.end\_button)

        vbox.addWidget(self.inference\_file\_button)

        vbox.addWidget(self.le)

        # vbox.addWidget(self.dialog)

        vbox.addWidget(self.quit\_button)

        # ------ Modification ------ #

        vbox.addWidget(self.capture\_button)

        # ------ Modification ------ #

        self.setLayout(vbox)

        self.setWindowTitle('Control Panel')

        self.setGeometry(100,100,200,200)

        self.show()

    def startCapture(self):

        if not self.capture:

            self.capture = QtCapture(0)

            self.end\_button.clicked.connect(self.capture.stop)

            # self.capture.setFPS(1)

            self.capture.setParent(self)

            self.capture.setWindowFlags(QtCore.Qt.Tool)

        self.capture.start()

        self.capture.show()

    def endCapture(self):

        self.capture.deleteLater()

        self.capture = None

    # ------ Modification ------ #

    def saveCapture(self):

        if self.capture:

            self.capture.capture()

    # ------ Modification ------ #

    def inf\_file(self):

        filename = QFileDialog.getOpenFileName()[0]

        self.le.setPixmap(QtGui.QPixmap(filename))

        im  = cv2.imread(filename)

        im = cv2.cvtColor(im, cv2.COLOR\_BGR2RGB)

        faces = extract\_faces(im)

        im\_mod = draw\_rectangles(im, faces, enlarge=True)

        cv2.imwrite('im\_mod.jpg', im\_mod)

        # except:

        #     print('wrong file')

    def picture(self):

        if not self.capture:

            self.capture = QtCapture(0)

            self.end\_button.clicked.connect(self.capture.stop)

        ret, frame = self.capture.cap.read()

        cv2.imwrite('img.jpg', frame)

        # My webcam yields frames in BGR format

        # frame = cv2.cvtColor(frame, cv2.COLOR\_BGR2RGB)

        # Face detection

        frame = draw\_rectangles(frame, extract\_faces(frame), enlarge=True)

        cv2.imwrite('img\_mod.jpg', frame)

if \_\_name\_\_ == '\_\_main\_\_':

    import sys

    app = QApplication(sys.argv)

    window = ControlWindow()

    sys.exit(app.exec\_())

yolo.py YOLO model

# YOLOv5 YOLO-specific modules

import argparse

import logging

import sys

from copy import deepcopy

from pathlib import Path

sys.path.append(Path(\_\_file\_\_).parent.parent.absolute().\_\_str\_\_())  # to run '$ python \*.py' files in subdirectories

logger = logging.getLogger(\_\_name\_\_)

from models.common import \*

from models.experimental import \*

from utils.autoanchor import check\_anchor\_order

from utils.general import make\_divisible, check\_file, set\_logging

from utils.torch\_utils import time\_synchronized, fuse\_conv\_and\_bn, model\_info, scale\_img, initialize\_weights, \

    select\_device, copy\_attr

try:

    import thop  # for FLOPS computation

except ImportError:

    thop = None

class Detect(nn.Module):

    stride = None  # strides computed during build

    onnx\_dynamic = False  # ONNX export parameter

    def \_\_init\_\_(self, nc=80, anchors=(), ch=(), inplace=True):  # detection layer

        super(Detect, self).\_\_init\_\_()

        self.nc = nc  # number of classes

        self.no = nc + 5  # number of outputs per anchor

        self.nl = len(anchors)  # number of detection layers

        self.na = len(anchors[0]) // 2  # number of anchors

        self.grid = [torch.zeros(1)] \* self.nl  # init grid

        a = torch.tensor(anchors).float().view(self.nl, -1, 2)

        self.register\_buffer('anchors', a)  # shape(nl,na,2)

        self.register\_buffer('anchor\_grid', a.clone().view(self.nl, 1, -1, 1, 1, 2))  # shape(nl,1,na,1,1,2)

        self.m = nn.ModuleList(nn.Conv2d(x, self.no \* self.na, 1) for x in ch)  # output conv

        self.inplace = inplace  # use in-place ops (e.g. slice assignment)

    def forward(self, x):

        # x = x.copy()  # for profiling

        z = []  # inference output

        for i in range(self.nl):

            x[i] = self.m[i](x[i])  # conv

            bs, \_, ny, nx = x[i].shape  # x(bs,255,20,20) to x(bs,3,20,20,85)

            x[i] = x[i].view(bs, self.na, self.no, ny, nx).permute(0, 1, 3, 4, 2).contiguous()

            if not self.training:  # inference

                if self.grid[i].shape[2:4] != x[i].shape[2:4] or self.onnx\_dynamic:

                    self.grid[i] = self.\_make\_grid(nx, ny).to(x[i].device)

                y = x[i].sigmoid()

                if self.inplace:

                    y[..., 0:2] = (y[..., 0:2] \* 2. - 0.5 + self.grid[i]) \* self.stride[i]  # xy

                    y[..., 2:4] = (y[..., 2:4] \* 2) \*\* 2 \* self.anchor\_grid[i]  # wh

                else:  # for YOLOv5 on AWS Inferentia https://github.com/ultralytics/yolov5/pull/2953

                    xy = (y[..., 0:2] \* 2. - 0.5 + self.grid[i]) \* self.stride[i]  # xy

                    wh = (y[..., 2:4] \* 2) \*\* 2 \* self.anchor\_grid[i].view(1, self.na, 1, 1, 2)  # wh

                    y = torch.cat((xy, wh, y[..., 4:]), -1)

                z.append(y.view(bs, -1, self.no))

        return x if self.training else (torch.cat(z, 1), x)

    @staticmethod

    def \_make\_grid(nx=20, ny=20):

        yv, xv = torch.meshgrid([torch.arange(ny), torch.arange(nx)])

        return torch.stack((xv, yv), 2).view((1, 1, ny, nx, 2)).float()

class Model(nn.Module):

    def \_\_init\_\_(self, cfg='yolov5s.yaml', ch=3, nc=None, anchors=None):  # model, input channels, number of classes

        super(Model, self).\_\_init\_\_()

        if isinstance(cfg, dict):

            self.yaml = cfg  # model dict

        else:  # is \*.yaml

            import yaml  # for torch hub

            self.yaml\_file = Path(cfg).name

            with open(cfg) as f:

                self.yaml = yaml.safe\_load(f)  # model dict

        # Define model

        ch = self.yaml['ch'] = self.yaml.get('ch', ch)  # input channels

        if nc and nc != self.yaml['nc']:

            logger.info(f"Overriding model.yaml nc={self.yaml['nc']} with nc={nc}")

            self.yaml['nc'] = nc  # override yaml value

        if anchors:

            logger.info(f'Overriding model.yaml anchors with anchors={anchors}')

            self.yaml['anchors'] = round(anchors)  # override yaml value

        self.model, self.save = parse\_model(deepcopy(self.yaml), ch=[ch])  # model, savelist

        self.names = [str(i) for i in range(self.yaml['nc'])]  # default names

        self.inplace = self.yaml.get('inplace', True)

        # logger.info([x.shape for x in self.forward(torch.zeros(1, ch, 64, 64))])

        # Build strides, anchors

        m = self.model[-1]  # Detect()

        if isinstance(m, Detect):

            s = 256  # 2x min stride

            m.inplace = self.inplace

            m.stride = torch.tensor([s / x.shape[-2] for x in self.forward(torch.zeros(1, ch, s, s))])  # forward

            m.anchors /= m.stride.view(-1, 1, 1)

            check\_anchor\_order(m)

            self.stride = m.stride

            self.\_initialize\_biases()  # only run once

            # logger.info('Strides: %s' % m.stride.tolist())

        # Init weights, biases

        initialize\_weights(self)

        self.info()

        logger.info('')

    def forward(self, x, augment=False, profile=False):

        if augment:

            return self.forward\_augment(x)  # augmented inference, None

        else:

            return self.forward\_once(x, profile)  # single-scale inference, train

    def forward\_augment(self, x):

        img\_size = x.shape[-2:]  # height, width

        s = [1, 0.83, 0.67]  # scales

        f = [None, 3, None]  # flips (2-ud, 3-lr)

        y = []  # outputs

        for si, fi in zip(s, f):

            xi = scale\_img(x.flip(fi) if fi else x, si, gs=int(self.stride.max()))

            yi = self.forward\_once(xi)[0]  # forward

            # cv2.imwrite(f'img\_{si}.jpg', 255 \* xi[0].cpu().numpy().transpose((1, 2, 0))[:, :, ::-1])  # save

            yi = self.\_descale\_pred(yi, fi, si, img\_size)

            y.append(yi)

        return torch.cat(y, 1), None  # augmented inference, train

    def forward\_once(self, x, profile=False):

        y, dt = [], []  # outputs

        for m in self.model:

            if m.f != -1:  # if not from previous layer

                x = y[m.f] if isinstance(m.f, int) else [x if j == -1 else y[j] for j in m.f]  # from earlier layers

            if profile:

                o = thop.profile(m, inputs=(x,), verbose=False)[0] / 1E9 \* 2 if thop else 0  # FLOPS

                t = time\_synchronized()

                for \_ in range(10):

                    \_ = m(x)

                dt.append((time\_synchronized() - t) \* 100)

                if m == self.model[0]:

                    logger.info(f"{'time (ms)':>10s} {'GFLOPS':>10s} {'params':>10s}  {'module'}")

                logger.info(f'{dt[-1]:10.2f} {o:10.2f} {m.np:10.0f}  {m.type}')

            x = m(x)  # run

            y.append(x if m.i in self.save else None)  # save output

        if profile:

            logger.info('%.1fms total' % sum(dt))

        return x

    def \_descale\_pred(self, p, flips, scale, img\_size):

        # de-scale predictions following augmented inference (inverse operation)

        if self.inplace:

            p[..., :4] /= scale  # de-scale

            if flips == 2:

                p[..., 1] = img\_size[0] - p[..., 1]  # de-flip ud

            elif flips == 3:

                p[..., 0] = img\_size[1] - p[..., 0]  # de-flip lr

        else:

            x, y, wh = p[..., 0:1] / scale, p[..., 1:2] / scale, p[..., 2:4] / scale  # de-scale

            if flips == 2:

                y = img\_size[0] - y  # de-flip ud

            elif flips == 3:

                x = img\_size[1] - x  # de-flip lr

            p = torch.cat((x, y, wh, p[..., 4:]), -1)

        return p

    def \_initialize\_biases(self, cf=None):  # initialize biases into Detect(), cf is class frequency

        # https://arxiv.org/abs/1708.02002 section 3.3

        # cf = torch.bincount(torch.tensor(np.concatenate(dataset.labels, 0)[:, 0]).long(), minlength=nc) + 1.

        m = self.model[-1]  # Detect() module

        for mi, s in zip(m.m, m.stride):  # from

            b = mi.bias.view(m.na, -1)  # conv.bias(255) to (3,85)

            b.data[:, 4] += math.log(8 / (640 / s) \*\* 2)  # obj (8 objects per 640 image)

            b.data[:, 5:] += math.log(0.6 / (m.nc - 0.99)) if cf is None else torch.log(cf / cf.sum())  # cls

            mi.bias = torch.nn.Parameter(b.view(-1), requires\_grad=True)

    def \_print\_biases(self):

        m = self.model[-1]  # Detect() module

        for mi in m.m:  # from

            b = mi.bias.detach().view(m.na, -1).T  # conv.bias(255) to (3,85)

            logger.info(

                ('%6g Conv2d.bias:' + '%10.3g' \* 6) % (mi.weight.shape[1], \*b[:5].mean(1).tolist(), b[5:].mean()))

    # def \_print\_weights(self):

    #     for m in self.model.modules():

    #         if type(m) is Bottleneck:

    #             logger.info('%10.3g' % (m.w.detach().sigmoid() \* 2))  # shortcut weights

    def fuse(self):  # fuse model Conv2d() + BatchNorm2d() layers

        logger.info('Fusing layers... ')

        for m in self.model.modules():

            if type(m) is Conv and hasattr(m, 'bn'):

                m.conv = fuse\_conv\_and\_bn(m.conv, m.bn)  # update conv

                delattr(m, 'bn')  # remove batchnorm

                m.forward = m.fuseforward  # update forward

        self.info()

        return self

    def nms(self, mode=True):  # add or remove NMS module

        present = type(self.model[-1]) is NMS  # last layer is NMS

        if mode and not present:

            logger.info('Adding NMS... ')

            m = NMS()  # module

            m.f = -1  # from

            m.i = self.model[-1].i + 1  # index

            self.model.add\_module(name='%s' % m.i, module=m)  # add

            self.eval()

        elif not mode and present:

            logger.info('Removing NMS... ')

            self.model = self.model[:-1]  # remove

        return self

    def autoshape(self):  # add autoShape module

        logger.info('Adding autoShape... ')

        m = autoShape(self)  # wrap model

        copy\_attr(m, self, include=('yaml', 'nc', 'hyp', 'names', 'stride'), exclude=())  # copy attributes

        return m

    def info(self, verbose=False, img\_size=640):  # print model information

        model\_info(self, verbose, img\_size)

def parse\_model(d, ch):  # model\_dict, input\_channels(3)

    logger.info('\n%3s%18s%3s%10s  %-40s%-30s' % ('', 'from', 'n', 'params', 'module', 'arguments'))

    anchors, nc, gd, gw = d['anchors'], d['nc'], d['depth\_multiple'], d['width\_multiple']

    na = (len(anchors[0]) // 2) if isinstance(anchors, list) else anchors  # number of anchors

    no = na \* (nc + 5)  # number of outputs = anchors \* (classes + 5)

    layers, save, c2 = [], [], ch[-1]  # layers, savelist, ch out

    for i, (f, n, m, args) in enumerate(d['backbone'] + d['head']):  # from, number, module, args

        m = eval(m) if isinstance(m, str) else m  # eval strings

        for j, a in enumerate(args):

            try:

                args[j] = eval(a) if isinstance(a, str) else a  # eval strings

            except:

                pass

        n = max(round(n \* gd), 1) if n > 1 else n  # depth gain

        if m in [Conv, GhostConv, Bottleneck, GhostBottleneck, SPP, DWConv, MixConv2d, Focus, CrossConv, BottleneckCSP,

                 C3, C3TR]:

            c1, c2 = ch[f], args[0]

            if c2 != no:  # if not output

                c2 = make\_divisible(c2 \* gw, 8)

            args = [c1, c2, \*args[1:]]

            if m in [BottleneckCSP, C3, C3TR]:

                args.insert(2, n)  # number of repeats

                n = 1

        elif m is nn.BatchNorm2d:

            args = [ch[f]]

        elif m is Concat:

            c2 = sum([ch[x] for x in f])

        elif m is Detect:

            args.append([ch[x] for x in f])

            if isinstance(args[1], int):  # number of anchors

                args[1] = [list(range(args[1] \* 2))] \* len(f)

        elif m is Contract:

            c2 = ch[f] \* args[0] \*\* 2

        elif m is Expand:

            c2 = ch[f] // args[0] \*\* 2

        else:

            c2 = ch[f]

        m\_ = nn.Sequential(\*[m(\*args) for \_ in range(n)]) if n > 1 else m(\*args)  # module

        t = str(m)[8:-2].replace('\_\_main\_\_.', '')  # module type

        np = sum([x.numel() for x in m\_.parameters()])  # number params

        m\_.i, m\_.f, m\_.type, m\_.np = i, f, t, np  # attach index, 'from' index, type, number params

        logger.info('%3s%18s%3s%10.0f  %-40s%-30s' % (i, f, n, np, t, args))  # print

        save.extend(x % i for x in ([f] if isinstance(f, int) else f) if x != -1)  # append to savelist

        layers.append(m\_)

        if i == 0:

            ch = []

        ch.append(c2)

    return nn.Sequential(\*layers), sorted(save)

if \_\_name\_\_ == '\_\_main\_\_':

    parser = argparse.ArgumentParser()

    parser.add\_argument('--cfg', type=str, default='yolov5s.yaml', help='model.yaml')

    parser.add\_argument('--device', default='', help='cuda device, i.e. 0 or 0,1,2,3 or cpu')

    opt = parser.parse\_args()

    opt.cfg = check\_file(opt.cfg)  # check file

    set\_logging()

    device = select\_device(opt.device)

    # Create model

    model = Model(opt.cfg).to(device)

    model.train()

mtcnn.py MTCNN

import torch

from torch import nn

import numpy as np

import os

from .utils.detect\_face import detect\_face, extract\_face

class PNet(nn.Module):

    """MTCNN PNet.

    Keyword Arguments:

        pretrained {bool} -- Whether or not to load saved pretrained weights (default: {True})

    """

    def \_\_init\_\_(self, pretrained=True):

        super().\_\_init\_\_()

        self.conv1 = nn.Conv2d(3, 10, kernel\_size=3)

        self.prelu1 = nn.PReLU(10)

        self.pool1 = nn.MaxPool2d(2, 2, ceil\_mode=True)

        self.conv2 = nn.Conv2d(10, 16, kernel\_size=3)

        self.prelu2 = nn.PReLU(16)

        self.conv3 = nn.Conv2d(16, 32, kernel\_size=3)

        self.prelu3 = nn.PReLU(32)

        self.conv4\_1 = nn.Conv2d(32, 2, kernel\_size=1)

        self.softmax4\_1 = nn.Softmax(dim=1)

        self.conv4\_2 = nn.Conv2d(32, 4, kernel\_size=1)

        self.training = False

        if pretrained:

            state\_dict\_path = os.path.join(os.path.dirname(\_\_file\_\_), '../data/pnet.pt')

            state\_dict = torch.load(state\_dict\_path)

            self.load\_state\_dict(state\_dict)

    def forward(self, x):

        x = self.conv1(x)

        x = self.prelu1(x)

        x = self.pool1(x)

        x = self.conv2(x)

        x = self.prelu2(x)

        x = self.conv3(x)

        x = self.prelu3(x)

        a = self.conv4\_1(x)

        a = self.softmax4\_1(a)

        b = self.conv4\_2(x)

        return b, a

class RNet(nn.Module):

    """MTCNN RNet.

    Keyword Arguments:

        pretrained {bool} -- Whether or not to load saved pretrained weights (default: {True})

    """

    def \_\_init\_\_(self, pretrained=True):

        super().\_\_init\_\_()

        self.conv1 = nn.Conv2d(3, 28, kernel\_size=3)

        self.prelu1 = nn.PReLU(28)

        self.pool1 = nn.MaxPool2d(3, 2, ceil\_mode=True)

        self.conv2 = nn.Conv2d(28, 48, kernel\_size=3)

        self.prelu2 = nn.PReLU(48)

        self.pool2 = nn.MaxPool2d(3, 2, ceil\_mode=True)

        self.conv3 = nn.Conv2d(48, 64, kernel\_size=2)

        self.prelu3 = nn.PReLU(64)

        self.dense4 = nn.Linear(576, 128)

        self.prelu4 = nn.PReLU(128)

        self.dense5\_1 = nn.Linear(128, 2)

        self.softmax5\_1 = nn.Softmax(dim=1)

        self.dense5\_2 = nn.Linear(128, 4)

        self.training = False

        if pretrained:

            state\_dict\_path = os.path.join(os.path.dirname(\_\_file\_\_), '../data/rnet.pt')

            state\_dict = torch.load(state\_dict\_path)

            self.load\_state\_dict(state\_dict)

    def forward(self, x):

        x = self.conv1(x)

        x = self.prelu1(x)

        x = self.pool1(x)

        x = self.conv2(x)

        x = self.prelu2(x)

        x = self.pool2(x)

        x = self.conv3(x)

        x = self.prelu3(x)

        x = x.permute(0, 3, 2, 1).contiguous()

        x = self.dense4(x.view(x.shape[0], -1))

        x = self.prelu4(x)

        a = self.dense5\_1(x)

        a = self.softmax5\_1(a)

        b = self.dense5\_2(x)

        return b, a

class ONet(nn.Module):

    """MTCNN ONet.

    Keyword Arguments:

        pretrained {bool} -- Whether or not to load saved pretrained weights (default: {True})

    """

    def \_\_init\_\_(self, pretrained=True):

        super().\_\_init\_\_()

        self.conv1 = nn.Conv2d(3, 32, kernel\_size=3)

        self.prelu1 = nn.PReLU(32)

        self.pool1 = nn.MaxPool2d(3, 2, ceil\_mode=True)

        self.conv2 = nn.Conv2d(32, 64, kernel\_size=3)

        self.prelu2 = nn.PReLU(64)

        self.pool2 = nn.MaxPool2d(3, 2, ceil\_mode=True)

        self.conv3 = nn.Conv2d(64, 64, kernel\_size=3)

        self.prelu3 = nn.PReLU(64)

        self.pool3 = nn.MaxPool2d(2, 2, ceil\_mode=True)

        self.conv4 = nn.Conv2d(64, 128, kernel\_size=2)

        self.prelu4 = nn.PReLU(128)

        self.dense5 = nn.Linear(1152, 256)

        self.prelu5 = nn.PReLU(256)

        self.dense6\_1 = nn.Linear(256, 2)

        self.softmax6\_1 = nn.Softmax(dim=1)

        self.dense6\_2 = nn.Linear(256, 4)

        self.dense6\_3 = nn.Linear(256, 10)

        self.training = False

        if pretrained:

            state\_dict\_path = os.path.join(os.path.dirname(\_\_file\_\_), '../data/onet.pt')

            state\_dict = torch.load(state\_dict\_path)

            self.load\_state\_dict(state\_dict)

    def forward(self, x):

        x = self.conv1(x)

        x = self.prelu1(x)

        x = self.pool1(x)

        x = self.conv2(x)

        x = self.prelu2(x)

        x = self.pool2(x)

        x = self.conv3(x)

        x = self.prelu3(x)

        x = self.pool3(x)

        x = self.conv4(x)

        x = self.prelu4(x)

        x = x.permute(0, 3, 2, 1).contiguous()

        x = self.dense5(x.view(x.shape[0], -1))

        x = self.prelu5(x)

        a = self.dense6\_1(x)

        a = self.softmax6\_1(a)

        b = self.dense6\_2(x)

        c = self.dense6\_3(x)

        return b, c, a

class MTCNN(nn.Module):

    """MTCNN face detection module.

    This class loads pretrained P-, R-, and O-nets and returns images cropped to include the face

    only, given raw input images of one of the following types:

        - PIL image or list of PIL images

        - numpy.ndarray (uint8) representing either a single image (3D) or a batch of images (4D).

    Cropped faces can optionally be saved to file

    also.

    Keyword Arguments:

        image\_size {int} -- Output image size in pixels. The image will be square. (default: {160})

        margin {int} -- Margin to add to bounding box, in terms of pixels in the final image.

            Note that the application of the margin differs slightly from the davidsandberg/facenet

            repo, which applies the margin to the original image before resizing, making the margin

            dependent on the original image size (this is a bug in davidsandberg/facenet).

            (default: {0})

        min\_face\_size {int} -- Minimum face size to search for. (default: {20})

        thresholds {list} -- MTCNN face detection thresholds (default: {[0.6, 0.7, 0.7]})

        factor {float} -- Factor used to create a scaling pyramid of face sizes. (default: {0.709})

        post\_process {bool} -- Whether or not to post process images tensors before returning.

            (default: {True})

        select\_largest {bool} -- If True, if multiple faces are detected, the largest is returned.

            If False, the face with the highest detection probability is returned.

            (default: {True})

        selection\_method {string} -- Which heuristic to use for selection. Default None. If

            specified, will override select\_largest:

                    "probability": highest probability selected

                    "largest": largest box selected

                    "largest\_over\_threshold": largest box over a certain probability selected

                    "center\_weighted\_size": box size minus weighted squared offset from image center

                (default: {None})

        keep\_all {bool} -- If True, all detected faces are returned, in the order dictated by the

            select\_largest parameter. If a save\_path is specified, the first face is saved to that

            path and the remaining faces are saved to <save\_path>1, <save\_path>2 etc.

            (default: {False})

        device {torch.device} -- The device on which to run neural net passes. Image tensors and

            models are copied to this device before running forward passes. (default: {None})

    """

    def \_\_init\_\_(

        self, image\_size=160, margin=0, min\_face\_size=20,

        thresholds=[0.6, 0.7, 0.7], factor=0.709, post\_process=True,

        select\_largest=True, selection\_method=None, keep\_all=False, device=None

    ):

        super().\_\_init\_\_()

        self.image\_size = image\_size

        self.margin = margin

        self.min\_face\_size = min\_face\_size

        self.thresholds = thresholds

        self.factor = factor

        self.post\_process = post\_process

        self.select\_largest = select\_largest

        self.keep\_all = keep\_all

        self.selection\_method = selection\_method

        self.pnet = PNet()

        self.rnet = RNet()

        self.onet = ONet()

        self.device = torch.device('cpu')

        if device is not None:

            self.device = device

            self.to(device)

        if not self.selection\_method:

            self.selection\_method = 'largest' if self.select\_largest else 'probability'

    def forward(self, img, save\_path=None, return\_prob=False):

        """Run MTCNN face detection on a PIL image or numpy array. This method performs both

        detection and extraction of faces, returning tensors representing detected faces rather

        than the bounding boxes. To access bounding boxes, see the MTCNN.detect() method below.

        Arguments:

            img {PIL.Image, np.ndarray, or list} -- A PIL image, np.ndarray, torch.Tensor, or list.

        Keyword Arguments:

            save\_path {str} -- An optional save path for the cropped image. Note that when

                self.post\_process=True, although the returned tensor is post processed, the saved

                face image is not, so it is a true representation of the face in the input image.

                If `img` is a list of images, `save\_path` should be a list of equal length.

                (default: {None})

            return\_prob {bool} -- Whether or not to return the detection probability.

                (default: {False})

        Returns:

            Union[torch.Tensor, tuple(torch.tensor, float)] -- If detected, cropped image of a face

                with dimensions 3 x image\_size x image\_size. Optionally, the probability that a

                face was detected. If self.keep\_all is True, n detected faces are returned in an

                n x 3 x image\_size x image\_size tensor with an optional list of detection

                probabilities. If `img` is a list of images, the item(s) returned have an extra

                dimension (batch) as the first dimension.

        Example:

        >>> from facenet\_pytorch import MTCNN

        >>> mtcnn = MTCNN()

        >>> face\_tensor, prob = mtcnn(img, save\_path='face.png', return\_prob=True)

        """

        # Detect faces

        batch\_boxes, batch\_probs, batch\_points = self.detect(img, landmarks=True)

        # Select faces

        if not self.keep\_all:

            batch\_boxes, batch\_probs, batch\_points = self.select\_boxes(

                batch\_boxes, batch\_probs, batch\_points, img, method=self.selection\_method

            )

        # Extract faces

        faces = self.extract(img, batch\_boxes, save\_path)

        if return\_prob:

            return faces, batch\_probs

        else:

            return faces

    def detect(self, img, landmarks=False):

        """Detect all faces in PIL image and return bounding boxes and optional facial landmarks.

        This method is used by the forward method and is also useful for face detection tasks

        that require lower-level handling of bounding boxes and facial landmarks (e.g., face

        tracking). The functionality of the forward function can be emulated by using this method

        followed by the extract\_face() function.

        Arguments:

            img {PIL.Image, np.ndarray, or list} -- A PIL image, np.ndarray, torch.Tensor, or list.

        Keyword Arguments:

            landmarks {bool} -- Whether to return facial landmarks in addition to bounding boxes.

                (default: {False})

        Returns:

            tuple(numpy.ndarray, list) -- For N detected faces, a tuple containing an

                Nx4 array of bounding boxes and a length N list of detection probabilities.

                Returned boxes will be sorted in descending order by detection probability if

                self.select\_largest=False, otherwise the largest face will be returned first.

                If `img` is a list of images, the items returned have an extra dimension

                (batch) as the first dimension. Optionally, a third item, the facial landmarks,

                are returned if `landmarks=True`.

        Example:

        >>> from PIL import Image, ImageDraw

        >>> from facenet\_pytorch import MTCNN, extract\_face

        >>> mtcnn = MTCNN(keep\_all=True)

        >>> boxes, probs, points = mtcnn.detect(img, landmarks=True)

        >>> # Draw boxes and save faces

        >>> img\_draw = img.copy()

        >>> draw = ImageDraw.Draw(img\_draw)

        >>> for i, (box, point) in enumerate(zip(boxes, points)):

        ...     draw.rectangle(box.tolist(), width=5)

        ...     for p in point:

        ...         draw.rectangle((p - 10).tolist() + (p + 10).tolist(), width=10)

        ...     extract\_face(img, box, save\_path='detected\_face\_{}.png'.format(i))

        >>> img\_draw.save('annotated\_faces.png')

        """

        with torch.no\_grad():

            batch\_boxes, batch\_points = detect\_face(

                img, self.min\_face\_size,

                self.pnet, self.rnet, self.onet,

                self.thresholds, self.factor,

                self.device

            )

        boxes, probs, points = [], [], []

        for box, point in zip(batch\_boxes, batch\_points):

            box = np.array(box)

            point = np.array(point)

            if len(box) == 0:

                boxes.append(None)

                probs.append([None])

                points.append(None)

            elif self.select\_largest:

                box\_order = np.argsort((box[:, 2] - box[:, 0]) \* (box[:, 3] - box[:, 1]))[::-1]

                box = box[box\_order]

                point = point[box\_order]

                boxes.append(box[:, :4])

                probs.append(box[:, 4])

                points.append(point)

            else:

                boxes.append(box[:, :4])

                probs.append(box[:, 4])

                points.append(point)

        boxes = np.array(boxes)

        probs = np.array(probs)

        points = np.array(points)

        if (

            not isinstance(img, (list, tuple)) and

            not (isinstance(img, np.ndarray) and len(img.shape) == 4) and

            not (isinstance(img, torch.Tensor) and len(img.shape) == 4)

        ):

            boxes = boxes[0]

            probs = probs[0]

            points = points[0]

        if landmarks:

            return boxes, probs, points

        return boxes, probs

    def select\_boxes(

        self, all\_boxes, all\_probs, all\_points, imgs, method='probability', threshold=0.9,

        center\_weight=2.0

    ):

        """Selects a single box from multiple for a given image using one of multiple heuristics.

        Arguments:

                all\_boxes {np.ndarray} -- Ix0 ndarray where each element is a Nx4 ndarry of

                    bounding boxes for N detected faces in I images (output from self.detect).

                all\_probs {np.ndarray} -- Ix0 ndarray where each element is a Nx0 ndarry of

                    probabilities for N detected faces in I images (output from self.detect).

                all\_points {np.ndarray} -- Ix0 ndarray where each element is a Nx5x2 array of

                    points for N detected faces. (output from self.detect).

                imgs {PIL.Image, np.ndarray, or list} -- A PIL image, np.ndarray, torch.Tensor, or list.

        Keyword Arguments:

                method {str} -- Which heuristic to use for selection:

                    "probability": highest probability selected

                    "largest": largest box selected

                    "largest\_over\_theshold": largest box over a certain probability selected

                    "center\_weighted\_size": box size minus weighted squared offset from image center

                    (default: {'probability'})

                threshold {float} -- theshold for "largest\_over\_threshold" method. (default: {0.9})

                center\_weight {float} -- weight for squared offset in center weighted size method.

                    (default: {2.0})

        Returns:

                tuple(numpy.ndarray, numpy.ndarray, numpy.ndarray) -- nx4 ndarray of bounding boxes

                    for n images. Ix0 array of probabilities for each box, array of landmark points.

        """

        #copying batch detection from extract, but would be easier to ensure detect creates consistent output.

        batch\_mode = True

        if (

                not isinstance(imgs, (list, tuple)) and

                not (isinstance(imgs, np.ndarray) and len(imgs.shape) == 4) and

                not (isinstance(imgs, torch.Tensor) and len(imgs.shape) == 4)

        ):

            imgs = [imgs]

            all\_boxes = [all\_boxes]

            all\_probs = [all\_probs]

            all\_points = [all\_points]

            batch\_mode = False

        selected\_boxes, selected\_probs, selected\_points = [], [], []

        for boxes, points, probs, img in zip(all\_boxes, all\_points, all\_probs, imgs):

            if boxes is None:

                selected\_boxes.append(None)

                selected\_probs.append([None])

                selected\_points.append(None)

                continue

            # If at least 1 box found

            boxes = np.array(boxes)

            probs = np.array(probs)

            points = np.array(points)

            if method == 'largest':

                box\_order = np.argsort((boxes[:, 2] - boxes[:, 0]) \* (boxes[:, 3] - boxes[:, 1]))[::-1]

            elif method == 'probability':

                box\_order = np.argsort(probs)[::-1]

            elif method == 'center\_weighted\_size':

                box\_sizes = (boxes[:, 2] - boxes[:, 0]) \* (boxes[:, 3] - boxes[:, 1])

                img\_center = (img.width / 2, img.height/2)

                box\_centers = np.array(list(zip((boxes[:, 0] + boxes[:, 2]) / 2, (boxes[:, 1] + boxes[:, 3]) / 2)))

                offsets = box\_centers - img\_center

                offset\_dist\_squared = np.sum(np.power(offsets, 2.0), 1)

                box\_order = np.argsort(box\_sizes - offset\_dist\_squared \* center\_weight)[::-1]

            elif method == 'largest\_over\_threshold':

                box\_mask = probs > threshold

                boxes = boxes[box\_mask]

                box\_order = np.argsort((boxes[:, 2] - boxes[:, 0]) \* (boxes[:, 3] - boxes[:, 1]))[::-1]

                if sum(box\_mask) == 0:

                    selected\_boxes.append(None)

                    selected\_probs.append([None])

                    selected\_points.append(None)

                    continue

            box = boxes[box\_order][[0]]

            prob = probs[box\_order][[0]]

            point = points[box\_order][[0]]

            selected\_boxes.append(box)

            selected\_probs.append(prob)

            selected\_points.append(point)

        if batch\_mode:

            selected\_boxes = np.array(selected\_boxes)

            selected\_probs = np.array(selected\_probs)

            selected\_points = np.array(selected\_points)

        else:

            selected\_boxes = selected\_boxes[0]

            selected\_probs = selected\_probs[0][0]

            selected\_points = selected\_points[0]

        return selected\_boxes, selected\_probs, selected\_points

    def extract(self, img, batch\_boxes, save\_path):

        # Determine if a batch or single image was passed

        batch\_mode = True

        if (

                not isinstance(img, (list, tuple)) and

                not (isinstance(img, np.ndarray) and len(img.shape) == 4) and

                not (isinstance(img, torch.Tensor) and len(img.shape) == 4)

        ):

            img = [img]

            batch\_boxes = [batch\_boxes]

            batch\_mode = False

        # Parse save path(s)

        if save\_path is not None:

            if isinstance(save\_path, str):

                save\_path = [save\_path]

        else:

            save\_path = [None for \_ in range(len(img))]

        # Process all bounding boxes

        faces = []

        for im, box\_im, path\_im in zip(img, batch\_boxes, save\_path):

            if box\_im is None:

                faces.append(None)

                continue

            if not self.keep\_all:

                box\_im = box\_im[[0]]

            faces\_im = []

            for i, box in enumerate(box\_im):

                face\_path = path\_im

                if path\_im is not None and i > 0:

                    save\_name, ext = os.path.splitext(path\_im)

                    face\_path = save\_name + '\_' + str(i + 1) + ext

                face = extract\_face(im, box, self.image\_size, self.margin, face\_path)

                if self.post\_process:

                    face = fixed\_image\_standardization(face)

                faces\_im.append(face)

            if self.keep\_all:

                faces\_im = torch.stack(faces\_im)

            else:

                faces\_im = faces\_im[0]

            faces.append(faces\_im)

        if not batch\_mode:

            faces = faces[0]

        return faces

def fixed\_image\_standardization(image\_tensor):

    processed\_tensor = (image\_tensor - 127.5) / 128.0

    return processed\_tensor

def prewhiten(x):

    mean = x.mean()

    std = x.std()

    std\_adj = std.clamp(min=1.0/(float(x.numel())\*\*0.5))

    y = (x - mean) / std\_adj

    return y

autoencor.py Autoencor functions from YOLOv5.

# Auto-anchor utils

import numpy as np

import torch

import yaml

from tqdm import tqdm

from utils.general import colorstr

def check\_anchor\_order(m):

    # Check anchor order against stride order for YOLOv5 Detect() module m, and correct if necessary

    a = m.anchor\_grid.prod(-1).view(-1)  # anchor area

    da = a[-1] - a[0]  # delta a

    ds = m.stride[-1] - m.stride[0]  # delta s

    if da.sign() != ds.sign():  # same order

        print('Reversing anchor order')

        m.anchors[:] = m.anchors.flip(0)

        m.anchor\_grid[:] = m.anchor\_grid.flip(0)

def check\_anchors(dataset, model, thr=4.0, imgsz=640):

    # Check anchor fit to data, recompute if necessary

    prefix = colorstr('autoanchor: ')

    print(f'\n{prefix}Analyzing anchors... ', end='')

    m = model.module.model[-1] if hasattr(model, 'module') else model.model[-1]  # Detect()

    shapes = imgsz \* dataset.shapes / dataset.shapes.max(1, keepdims=True)

    scale = np.random.uniform(0.9, 1.1, size=(shapes.shape[0], 1))  # augment scale

    wh = torch.tensor(np.concatenate([l[:, 3:5] \* s for s, l in zip(shapes \* scale, dataset.labels)])).float()  # wh

    def metric(k):  # compute metric

        r = wh[:, None] / k[None]

        x = torch.min(r, 1. / r).min(2)[0]  # ratio metric

        best = x.max(1)[0]  # best\_x

        aat = (x > 1. / thr).float().sum(1).mean()  # anchors above threshold

        bpr = (best > 1. / thr).float().mean()  # best possible recall

        return bpr, aat

    anchors = m.anchor\_grid.clone().cpu().view(-1, 2)  # current anchors

    bpr, aat = metric(anchors)

    print(f'anchors/target = {aat:.2f}, Best Possible Recall (BPR) = {bpr:.4f}', end='')

    if bpr < 0.98:  # threshold to recompute

        print('. Attempting to improve anchors, please wait...')

        na = m.anchor\_grid.numel() // 2  # number of anchors

        try:

            anchors = kmean\_anchors(dataset, n=na, img\_size=imgsz, thr=thr, gen=1000, verbose=False)

        except Exception as e:

            print(f'{prefix}ERROR: {e}')

        new\_bpr = metric(anchors)[0]

        if new\_bpr > bpr:  # replace anchors

            anchors = torch.tensor(anchors, device=m.anchors.device).type\_as(m.anchors)

            m.anchor\_grid[:] = anchors.clone().view\_as(m.anchor\_grid)  # for inference

            m.anchors[:] = anchors.clone().view\_as(m.anchors) / m.stride.to(m.anchors.device).view(-1, 1, 1)  # loss

            check\_anchor\_order(m)

            print(f'{prefix}New anchors saved to model. Update model \*.yaml to use these anchors in the future.')

        else:

            print(f'{prefix}Original anchors better than new anchors. Proceeding with original anchors.')

    print('')  # newline

def kmean\_anchors(path='./data/coco128.yaml', n=9, img\_size=640, thr=4.0, gen=1000, verbose=True):

    """ Creates kmeans-evolved anchors from training dataset

        Arguments:

            path: path to dataset \*.yaml, or a loaded dataset

            n: number of anchors

            img\_size: image size used for training

            thr: anchor-label wh ratio threshold hyperparameter hyp['anchor\_t'] used for training, default=4.0

            gen: generations to evolve anchors using genetic algorithm

            verbose: print all results

        Return:

            k: kmeans evolved anchors

        Usage:

            from utils.autoanchor import \*; \_ = kmean\_anchors()

    """

    from scipy.cluster.vq import kmeans

    thr = 1. / thr

    prefix = colorstr('autoanchor: ')

    def metric(k, wh):  # compute metrics

        r = wh[:, None] / k[None]

        x = torch.min(r, 1. / r).min(2)[0]  # ratio metric

        # x = wh\_iou(wh, torch.tensor(k))  # iou metric

        return x, x.max(1)[0]  # x, best\_x

    def anchor\_fitness(k):  # mutation fitness

        \_, best = metric(torch.tensor(k, dtype=torch.float32), wh)

        return (best \* (best > thr).float()).mean()  # fitness

    def print\_results(k):

        k = k[np.argsort(k.prod(1))]  # sort small to large

        x, best = metric(k, wh0)

        bpr, aat = (best > thr).float().mean(), (x > thr).float().mean() \* n  # best possible recall, anch > thr

        print(f'{prefix}thr={thr:.2f}: {bpr:.4f} best possible recall, {aat:.2f} anchors past thr')

        print(f'{prefix}n={n}, img\_size={img\_size}, metric\_all={x.mean():.3f}/{best.mean():.3f}-mean/best, '

              f'past\_thr={x[x > thr].mean():.3f}-mean: ', end='')

        for i, x in enumerate(k):

            print('%i,%i' % (round(x[0]), round(x[1])), end=',  ' if i < len(k) - 1 else '\n')  # use in \*.cfg

        return k

    if isinstance(path, str):  # \*.yaml file

        with open(path) as f:

            data\_dict = yaml.safe\_load(f)  # model dict

        from utils.datasets import LoadImagesAndLabels

        dataset = LoadImagesAndLabels(data\_dict['train'], augment=True, rect=True)

    else:

        dataset = path  # dataset

    # Get label wh

    shapes = img\_size \* dataset.shapes / dataset.shapes.max(1, keepdims=True)

    wh0 = np.concatenate([l[:, 3:5] \* s for s, l in zip(shapes, dataset.labels)])  # wh

    # Filter

    i = (wh0 < 3.0).any(1).sum()

    if i:

        print(f'{prefix}WARNING: Extremely small objects found. {i} of {len(wh0)} labels are < 3 pixels in size.')

    wh = wh0[(wh0 >= 2.0).any(1)]  # filter > 2 pixels

    # wh = wh \* (np.random.rand(wh.shape[0], 1) \* 0.9 + 0.1)  # multiply by random scale 0-1

    # Kmeans calculation

    print(f'{prefix}Running kmeans for {n} anchors on {len(wh)} points...')

    s = wh.std(0)  # sigmas for whitening

    k, dist = kmeans(wh / s, n, iter=30)  # points, mean distance

    assert len(k) == n, print(f'{prefix}ERROR: scipy.cluster.vq.kmeans requested {n} points but returned only {len(k)}')

    k \*= s

    wh = torch.tensor(wh, dtype=torch.float32)  # filtered

    wh0 = torch.tensor(wh0, dtype=torch.float32)  # unfiltered

    k = print\_results(k)

    # Plot

    # k, d = [None] \* 20, [None] \* 20

    # for i in tqdm(range(1, 21)):

    #     k[i-1], d[i-1] = kmeans(wh / s, i)  # points, mean distance

    # fig, ax = plt.subplots(1, 2, figsize=(14, 7), tight\_layout=True)

    # ax = ax.ravel()

    # ax[0].plot(np.arange(1, 21), np.array(d) \*\* 2, marker='.')

    # fig, ax = plt.subplots(1, 2, figsize=(14, 7))  # plot wh

    # ax[0].hist(wh[wh[:, 0]<100, 0],400)

    # ax[1].hist(wh[wh[:, 1]<100, 1],400)

    # fig.savefig('wh.png', dpi=200)

    # Evolve

    npr = np.random

    f, sh, mp, s = anchor\_fitness(k), k.shape, 0.9, 0.1  # fitness, generations, mutation prob, sigma

    pbar = tqdm(range(gen), desc=f'{prefix}Evolving anchors with Genetic Algorithm:')  # progress bar

    for \_ in pbar:

        v = np.ones(sh)

        while (v == 1).all():  # mutate until a change occurs (prevent duplicates)

            v = ((npr.random(sh) < mp) \* npr.random() \* npr.randn(\*sh) \* s + 1).clip(0.3, 3.0)

        kg = (k.copy() \* v).clip(min=2.0)

        fg = anchor\_fitness(kg)

        if fg > f:

            f, k = fg, kg.copy()

            pbar.desc = f'{prefix}Evolving anchors with Genetic Algorithm: fitness = {f:.4f}'

            if verbose:

                print\_results(k)

    return print\_results(k)

yolo/utils/general.py data format changers and other utils for YOLO

# YOLOv5 general utils

import glob

import logging

import math

import os

import platform

import random

import re

import subprocess

import time

from itertools import repeat

from multiprocessing.pool import ThreadPool

from pathlib import Path

import cv2

import numpy as np

import pandas as pd

import pkg\_resources as pkg

import torch

import torchvision

import yaml

from utils.google\_utils import gsutil\_getsize

from utils.metrics import fitness

from utils.torch\_utils import init\_torch\_seeds

# Settings

torch.set\_printoptions(linewidth=320, precision=5, profile='long')

np.set\_printoptions(linewidth=320, formatter={'float\_kind': '{:11.5g}'.format})  # format short g, %precision=5

pd.options.display.max\_columns = 10

cv2.setNumThreads(0)  # prevent OpenCV from multithreading (incompatible with PyTorch DataLoader)

os.environ['NUMEXPR\_MAX\_THREADS'] = str(min(os.cpu\_count(), 8))  # NumExpr max threads

def set\_logging(rank=-1, verbose=True):

    logging.basicConfig(

        format="%(message)s",

        level=logging.INFO if (verbose and rank in [-1, 0]) else logging.WARN)

def init\_seeds(seed=0):

    # Initialize random number generator (RNG) seeds

    random.seed(seed)

    np.random.seed(seed)

    init\_torch\_seeds(seed)

def get\_latest\_run(search\_dir='.'):

    # Return path to most recent 'last.pt' in /runs (i.e. to --resume from)

    last\_list = glob.glob(f'{search\_dir}/\*\*/last\*.pt', recursive=True)

    return max(last\_list, key=os.path.getctime) if last\_list else ''

def is\_docker():

    # Is environment a Docker container

    return Path('/workspace').exists()  # or Path('/.dockerenv').exists()

def is\_colab():

    # Is environment a Google Colab instance

    try:

        import google.colab

        return True

    except Exception as e:

        return False

def emojis(str=''):

    # Return platform-dependent emoji-safe version of string

    return str.encode().decode('ascii', 'ignore') if platform.system() == 'Windows' else str

def file\_size(file):

    # Return file size in MB

    return Path(file).stat().st\_size / 1e6

def check\_online():

    # Check internet connectivity

    import socket

    try:

        socket.create\_connection(("1.1.1.1", 443), 5)  # check host accesability

        return True

    except OSError:

        return False

def check\_git\_status():

    # Recommend 'git pull' if code is out of date

    print(colorstr('github: '), end='')

    try:

        assert Path('.git').exists(), 'skipping check (not a git repository)'

        assert not is\_docker(), 'skipping check (Docker image)'

        assert check\_online(), 'skipping check (offline)'

        cmd = 'git fetch && git config --get remote.origin.url'

        url = subprocess.check\_output(cmd, shell=True).decode().strip().rstrip('.git')  # github repo url

        branch = subprocess.check\_output('git rev-parse --abbrev-ref HEAD', shell=True).decode().strip()  # checked out

        n = int(subprocess.check\_output(f'git rev-list {branch}..origin/master --count', shell=True))  # commits behind

        if n > 0:

            s = f"⚠️ WARNING: code is out of date by {n} commit{'s' \* (n > 1)}. " \

                f"Use 'git pull' to update or 'git clone {url}' to download latest."

        else:

            s = f'up to date with {url} ✅'

        print(emojis(s))  # emoji-safe

    except Exception as e:

        print(e)

def check\_python(minimum='3.7.0', required=True):

    # Check current python version vs. required python version

    current = platform.python\_version()

    result = pkg.parse\_version(current) >= pkg.parse\_version(minimum)

    if required:

        assert result, f'Python {minimum} required by YOLOv5, but Python {current} is currently installed'

    return result

def check\_requirements(requirements='requirements.txt', exclude=()):

    # Check installed dependencies meet requirements (pass \*.txt file or list of packages)

    prefix = colorstr('red', 'bold', 'requirements:')

    check\_python()  # check python version

    if isinstance(requirements, (str, Path)):  # requirements.txt file

        file = Path(requirements)

        if not file.exists():

            print(f"{prefix} {file.resolve()} not found, check failed.")

            return

        requirements = [f'{x.name}{x.specifier}' for x in pkg.parse\_requirements(file.open()) if x.name not in exclude]

    else:  # list or tuple of packages

        requirements = [x for x in requirements if x not in exclude]

    n = 0  # number of packages updates

    for r in requirements:

        try:

            pkg.require(r)

        except Exception as e:  # DistributionNotFound or VersionConflict if requirements not met

            n += 1

            print(f"{prefix} {r} not found and is required by YOLOv5, attempting auto-update...")

            print(subprocess.check\_output(f"pip install '{r}'", shell=True).decode())

    if n:  # if packages updated

        source = file.resolve() if 'file' in locals() else requirements

        s = f"{prefix} {n} package{'s' \* (n > 1)} updated per {source}\n" \

            f"{prefix} ⚠️ {colorstr('bold', 'Restart runtime or rerun command for updates to take effect')}\n"

        print(emojis(s))  # emoji-safe

def check\_img\_size(img\_size, s=32):

    # Verify img\_size is a multiple of stride s

    new\_size = make\_divisible(img\_size, int(s))  # ceil gs-multiple

    if new\_size != img\_size:

        print('WARNING: --img-size %g must be multiple of max stride %g, updating to %g' % (img\_size, s, new\_size))

    return new\_size

def check\_imshow():

    # Check if environment supports image displays

    try:

        assert not is\_docker(), 'cv2.imshow() is disabled in Docker environments'

        assert not is\_colab(), 'cv2.imshow() is disabled in Google Colab environments'

        cv2.imshow('test', np.zeros((1, 1, 3)))

        cv2.waitKey(1)

        cv2.destroyAllWindows()

        cv2.waitKey(1)

        return True

    except Exception as e:

        print(f'WARNING: Environment does not support cv2.imshow() or PIL Image.show() image displays\n{e}')

        return False

def check\_file(file):

    # Search for file if not found

    if Path(file).is\_file() or file == '':

        return file

    else:

        files = glob.glob('./\*\*/' + file, recursive=True)  # find file

        assert len(files), f'File Not Found: {file}'  # assert file was found

        assert len(files) == 1, f"Multiple files match '{file}', specify exact path: {files}"  # assert unique

        return files[0]  # return file

def check\_dataset(dict):

    # Download dataset if not found locally

    val, s = dict.get('val'), dict.get('download')

    if val and len(val):

        val = [Path(x).resolve() for x in (val if isinstance(val, list) else [val])]  # val path

        if not all(x.exists() for x in val):

            print('\nWARNING: Dataset not found, nonexistent paths: %s' % [str(x) for x in val if not x.exists()])

            if s and len(s):  # download script

                if s.startswith('http') and s.endswith('.zip'):  # URL

                    f = Path(s).name  # filename

                    print(f'Downloading {s} ...')

                    torch.hub.download\_url\_to\_file(s, f)

                    r = os.system(f'unzip -q {f} -d ../ && rm {f}')  # unzip

                elif s.startswith('bash '):  # bash script

                    print(f'Running {s} ...')

                    r = os.system(s)

                else:  # python script

                    r = exec(s)  # return None

                print('Dataset autodownload %s\n' % ('success' if r in (0, None) else 'failure'))  # print result

            else:

                raise Exception('Dataset not found.')

def download(url, dir='.', unzip=True, delete=True, curl=False, threads=1):

    # Multi-threaded file download and unzip function

    def download\_one(url, dir):

        # Download 1 file

        f = dir / Path(url).name  # filename

        if not f.exists():

            print(f'Downloading {url} to {f}...')

            if curl:

                os.system(f"curl -L '{url}' -o '{f}' --retry 9 -C -")  # curl download, retry and resume on fail

            else:

                torch.hub.download\_url\_to\_file(url, f, progress=True)  # torch download

        if unzip and f.suffix in ('.zip', '.gz'):

            print(f'Unzipping {f}...')

            if f.suffix == '.zip':

                s = f'unzip -qo {f} -d {dir} && rm {f}'  # unzip -quiet -overwrite

            elif f.suffix == '.gz':

                s = f'tar xfz {f} --directory {f.parent}'  # unzip

            if delete:  # delete zip file after unzip

                s += f' && rm {f}'

            os.system(s)

    dir = Path(dir)

    dir.mkdir(parents=True, exist\_ok=True)  # make directory

    if threads > 1:

        pool = ThreadPool(threads)

        pool.imap(lambda x: download\_one(\*x), zip(url, repeat(dir)))  # multi-threaded

        pool.close()

        pool.join()

    else:

        for u in tuple(url) if isinstance(url, str) else url:

            download\_one(u, dir)

def make\_divisible(x, divisor):

    # Returns x evenly divisible by divisor

    return math.ceil(x / divisor) \* divisor

def clean\_str(s):

    # Cleans a string by replacing special characters with underscore \_

    return re.sub(pattern="[|@#!¡·$€%&()=?¿^\*;:,¨´><+]", repl="\_", string=s)

def one\_cycle(y1=0.0, y2=1.0, steps=100):

    # lambda function for sinusoidal ramp from y1 to y2

    return lambda x: ((1 - math.cos(x \* math.pi / steps)) / 2) \* (y2 - y1) + y1

def colorstr(\*input):

    # Colors a string https://en.wikipedia.org/wiki/ANSI\_escape\_code, i.e.  colorstr('blue', 'hello world')

    \*args, string = input if len(input) > 1 else ('blue', 'bold', input[0])  # color arguments, string

    colors = {'black': '\033[30m',  # basic colors

              'red': '\033[31m',

              'green': '\033[32m',

              'yellow': '\033[33m',

              'blue': '\033[34m',

              'magenta': '\033[35m',

              'cyan': '\033[36m',

              'white': '\033[37m',

              'bright\_black': '\033[90m',  # bright colors

              'bright\_red': '\033[91m',

              'bright\_green': '\033[92m',

              'bright\_yellow': '\033[93m',

              'bright\_blue': '\033[94m',

              'bright\_magenta': '\033[95m',

              'bright\_cyan': '\033[96m',

              'bright\_white': '\033[97m',

              'end': '\033[0m',  # misc

              'bold': '\033[1m',

              'underline': '\033[4m'}

    return ''.join(colors[x] for x in args) + f'{string}' + colors['end']

def labels\_to\_class\_weights(labels, nc=80):

    # Get class weights (inverse frequency) from training labels

    if labels[0] is None:  # no labels loaded

        return torch.Tensor()

    labels = np.concatenate(labels, 0)  # labels.shape = (866643, 5) for COCO

    classes = labels[:, 0].astype(np.int)  # labels = [class xywh]

    weights = np.bincount(classes, minlength=nc)  # occurrences per class

    # Prepend gridpoint count (for uCE training)

    # gpi = ((320 / 32 \* np.array([1, 2, 4])) \*\* 2 \* 3).sum()  # gridpoints per image

    # weights = np.hstack([gpi \* len(labels)  - weights.sum() \* 9, weights \* 9]) \*\* 0.5  # prepend gridpoints to start

    weights[weights == 0] = 1  # replace empty bins with 1

    weights = 1 / weights  # number of targets per class

    weights /= weights.sum()  # normalize

    return torch.from\_numpy(weights)

def labels\_to\_image\_weights(labels, nc=80, class\_weights=np.ones(80)):

    # Produces image weights based on class\_weights and image contents

    class\_counts = np.array([np.bincount(x[:, 0].astype(np.int), minlength=nc) for x in labels])

    image\_weights = (class\_weights.reshape(1, nc) \* class\_counts).sum(1)

    # index = random.choices(range(n), weights=image\_weights, k=1)  # weight image sample

    return image\_weights

def coco80\_to\_coco91\_class():  # converts 80-index (val2014) to 91-index (paper)

    # https://tech.amikelive.com/node-718/what-object-categories-labels-are-in-coco-dataset/

    # a = np.loadtxt('data/coco.names', dtype='str', delimiter='\n')

    # b = np.loadtxt('data/coco\_paper.names', dtype='str', delimiter='\n')

    # x1 = [list(a[i] == b).index(True) + 1 for i in range(80)]  # darknet to coco

    # x2 = [list(b[i] == a).index(True) if any(b[i] == a) else None for i in range(91)]  # coco to darknet

    x = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24, 25, 27, 28, 31, 32, 33, 34,

         35, 36, 37, 38, 39, 40, 41, 42, 43, 44, 46, 47, 48, 49, 50, 51, 52, 53, 54, 55, 56, 57, 58, 59, 60, 61, 62, 63,

         64, 65, 67, 70, 72, 73, 74, 75, 76, 77, 78, 79, 80, 81, 82, 84, 85, 86, 87, 88, 89, 90]

    return x

def xyxy2xywh(x):

    # Convert nx4 boxes from [x1, y1, x2, y2] to [x, y, w, h] where xy1=top-left, xy2=bottom-right

    y = x.clone() if isinstance(x, torch.Tensor) else np.copy(x)

    y[:, 0] = (x[:, 0] + x[:, 2]) / 2  # x center

    y[:, 1] = (x[:, 1] + x[:, 3]) / 2  # y center

    y[:, 2] = x[:, 2] - x[:, 0]  # width

    y[:, 3] = x[:, 3] - x[:, 1]  # height

    return y

def xywh2xyxy(x):

    # Convert nx4 boxes from [x, y, w, h] to [x1, y1, x2, y2] where xy1=top-left, xy2=bottom-right

    y = x.clone() if isinstance(x, torch.Tensor) else np.copy(x)

    y[:, 0] = x[:, 0] - x[:, 2] / 2  # top left x

    y[:, 1] = x[:, 1] - x[:, 3] / 2  # top left y

    y[:, 2] = x[:, 0] + x[:, 2] / 2  # bottom right x

    y[:, 3] = x[:, 1] + x[:, 3] / 2  # bottom right y

    return y

def xywhn2xyxy(x, w=640, h=640, padw=0, padh=0):

    # Convert nx4 boxes from [x, y, w, h] normalized to [x1, y1, x2, y2] where xy1=top-left, xy2=bottom-right

    y = x.clone() if isinstance(x, torch.Tensor) else np.copy(x)

    y[:, 0] = w \* (x[:, 0] - x[:, 2] / 2) + padw  # top left x

    y[:, 1] = h \* (x[:, 1] - x[:, 3] / 2) + padh  # top left y

    y[:, 2] = w \* (x[:, 0] + x[:, 2] / 2) + padw  # bottom right x

    y[:, 3] = h \* (x[:, 1] + x[:, 3] / 2) + padh  # bottom right y

    return y

def xyn2xy(x, w=640, h=640, padw=0, padh=0):

    # Convert normalized segments into pixel segments, shape (n,2)

    y = x.clone() if isinstance(x, torch.Tensor) else np.copy(x)

    y[:, 0] = w \* x[:, 0] + padw  # top left x

    y[:, 1] = h \* x[:, 1] + padh  # top left y

    return y

def segment2box(segment, width=640, height=640):

    # Convert 1 segment label to 1 box label, applying inside-image constraint, i.e. (xy1, xy2, ...) to (xyxy)

    x, y = segment.T  # segment xy

    inside = (x >= 0) & (y >= 0) & (x <= width) & (y <= height)

    x, y, = x[inside], y[inside]

    return np.array([x.min(), y.min(), x.max(), y.max()]) if any(x) else np.zeros((1, 4))  # xyxy

def segments2boxes(segments):

    # Convert segment labels to box labels, i.e. (cls, xy1, xy2, ...) to (cls, xywh)

    boxes = []

    for s in segments:

        x, y = s.T  # segment xy

        boxes.append([x.min(), y.min(), x.max(), y.max()])  # cls, xyxy

    return xyxy2xywh(np.array(boxes))  # cls, xywh

def resample\_segments(segments, n=1000):

    # Up-sample an (n,2) segment

    for i, s in enumerate(segments):

        x = np.linspace(0, len(s) - 1, n)

        xp = np.arange(len(s))

        segments[i] = np.concatenate([np.interp(x, xp, s[:, i]) for i in range(2)]).reshape(2, -1).T  # segment xy

    return segments

def scale\_coords(img1\_shape, coords, img0\_shape, ratio\_pad=None):

    # Rescale coords (xyxy) from img1\_shape to img0\_shape

    if ratio\_pad is None:  # calculate from img0\_shape

        gain = min(img1\_shape[0] / img0\_shape[0], img1\_shape[1] / img0\_shape[1])  # gain  = old / new

        pad = (img1\_shape[1] - img0\_shape[1] \* gain) / 2, (img1\_shape[0] - img0\_shape[0] \* gain) / 2  # wh padding

    else:

        gain = ratio\_pad[0][0]

        pad = ratio\_pad[1]

    coords[:, [0, 2]] -= pad[0]  # x padding

    coords[:, [1, 3]] -= pad[1]  # y padding

    coords[:, :4] /= gain

    clip\_coords(coords, img0\_shape)

    return coords

def clip\_coords(boxes, img\_shape):

    # Clip bounding xyxy bounding boxes to image shape (height, width)

    boxes[:, 0].clamp\_(0, img\_shape[1])  # x1

    boxes[:, 1].clamp\_(0, img\_shape[0])  # y1

    boxes[:, 2].clamp\_(0, img\_shape[1])  # x2

    boxes[:, 3].clamp\_(0, img\_shape[0])  # y2

def bbox\_iou(box1, box2, x1y1x2y2=True, GIoU=False, DIoU=False, CIoU=False, eps=1e-7):

    # Returns the IoU of box1 to box2. box1 is 4, box2 is nx4

    box2 = box2.T

    # Get the coordinates of bounding boxes

    if x1y1x2y2:  # x1, y1, x2, y2 = box1

        b1\_x1, b1\_y1, b1\_x2, b1\_y2 = box1[0], box1[1], box1[2], box1[3]

        b2\_x1, b2\_y1, b2\_x2, b2\_y2 = box2[0], box2[1], box2[2], box2[3]

    else:  # transform from xywh to xyxy

        b1\_x1, b1\_x2 = box1[0] - box1[2] / 2, box1[0] + box1[2] / 2

        b1\_y1, b1\_y2 = box1[1] - box1[3] / 2, box1[1] + box1[3] / 2

        b2\_x1, b2\_x2 = box2[0] - box2[2] / 2, box2[0] + box2[2] / 2

        b2\_y1, b2\_y2 = box2[1] - box2[3] / 2, box2[1] + box2[3] / 2

    # Intersection area

    inter = (torch.min(b1\_x2, b2\_x2) - torch.max(b1\_x1, b2\_x1)).clamp(0) \* \

            (torch.min(b1\_y2, b2\_y2) - torch.max(b1\_y1, b2\_y1)).clamp(0)

    # Union Area

    w1, h1 = b1\_x2 - b1\_x1, b1\_y2 - b1\_y1 + eps

    w2, h2 = b2\_x2 - b2\_x1, b2\_y2 - b2\_y1 + eps

    union = w1 \* h1 + w2 \* h2 - inter + eps

    iou = inter / union

    if GIoU or DIoU or CIoU:

        cw = torch.max(b1\_x2, b2\_x2) - torch.min(b1\_x1, b2\_x1)  # convex (smallest enclosing box) width

        ch = torch.max(b1\_y2, b2\_y2) - torch.min(b1\_y1, b2\_y1)  # convex height

        if CIoU or DIoU:  # Distance or Complete IoU https://arxiv.org/abs/1911.08287v1

            c2 = cw \*\* 2 + ch \*\* 2 + eps  # convex diagonal squared

            rho2 = ((b2\_x1 + b2\_x2 - b1\_x1 - b1\_x2) \*\* 2 +

                    (b2\_y1 + b2\_y2 - b1\_y1 - b1\_y2) \*\* 2) / 4  # center distance squared

            if DIoU:

                return iou - rho2 / c2  # DIoU

            elif CIoU:  # https://github.com/Zzh-tju/DIoU-SSD-pytorch/blob/master/utils/box/box\_utils.py#L47

                v = (4 / math.pi \*\* 2) \* torch.pow(torch.atan(w2 / h2) - torch.atan(w1 / h1), 2)

                with torch.no\_grad():

                    alpha = v / (v - iou + (1 + eps))

                return iou - (rho2 / c2 + v \* alpha)  # CIoU

        else:  # GIoU https://arxiv.org/pdf/1902.09630.pdf

            c\_area = cw \* ch + eps  # convex area

            return iou - (c\_area - union) / c\_area  # GIoU

    else:

        return iou  # IoU

def box\_iou(box1, box2):

    # https://github.com/pytorch/vision/blob/master/torchvision/ops/boxes.py

    """

    Return intersection-over-union (Jaccard index) of boxes.

    Both sets of boxes are expected to be in (x1, y1, x2, y2) format.

    Arguments:

        box1 (Tensor[N, 4])

        box2 (Tensor[M, 4])

    Returns:

        iou (Tensor[N, M]): the NxM matrix containing the pairwise

            IoU values for every element in boxes1 and boxes2

    """

    def box\_area(box):

        # box = 4xn

        return (box[2] - box[0]) \* (box[3] - box[1])

    area1 = box\_area(box1.T)

    area2 = box\_area(box2.T)

    # inter(N,M) = (rb(N,M,2) - lt(N,M,2)).clamp(0).prod(2)

    inter = (torch.min(box1[:, None, 2:], box2[:, 2:]) - torch.max(box1[:, None, :2], box2[:, :2])).clamp(0).prod(2)

    return inter / (area1[:, None] + area2 - inter)  # iou = inter / (area1 + area2 - inter)

def wh\_iou(wh1, wh2):

    # Returns the nxm IoU matrix. wh1 is nx2, wh2 is mx2

    wh1 = wh1[:, None]  # [N,1,2]

    wh2 = wh2[None]  # [1,M,2]

    inter = torch.min(wh1, wh2).prod(2)  # [N,M]

    return inter / (wh1.prod(2) + wh2.prod(2) - inter)  # iou = inter / (area1 + area2 - inter)

def non\_max\_suppression(prediction, conf\_thres=0.25, iou\_thres=0.45, classes=None, agnostic=False, multi\_label=False,

                        labels=()):

    """Runs Non-Maximum Suppression (NMS) on inference results

    Returns:

         list of detections, on (n,6) tensor per image [xyxy, conf, cls]

    """

    nc = prediction.shape[2] - 5  # number of classes

    xc = prediction[..., 4] > conf\_thres  # candidates

    # Checks

    assert 0 <= conf\_thres <= 1, f'Invalid Confidence threshold {conf\_thres}, valid values are between 0.0 and 1.0'

    assert 0 <= iou\_thres <= 1, f'Invalid IoU {iou\_thres}, valid values are between 0.0 and 1.0'

    # Settings

    min\_wh, max\_wh = 2, 4096  # (pixels) minimum and maximum box width and height

    max\_det = 300  # maximum number of detections per image

    max\_nms = 30000  # maximum number of boxes into torchvision.ops.nms()

    time\_limit = 10.0  # seconds to quit after

    redundant = True  # require redundant detections

    multi\_label &= nc > 1  # multiple labels per box (adds 0.5ms/img)

    merge = False  # use merge-NMS

    t = time.time()

    output = [torch.zeros((0, 6), device=prediction.device)] \* prediction.shape[0]

    for xi, x in enumerate(prediction):  # image index, image inference

        # Apply constraints

        # x[((x[..., 2:4] < min\_wh) | (x[..., 2:4] > max\_wh)).any(1), 4] = 0  # width-height

        x = x[xc[xi]]  # confidence

        # Cat apriori labels if autolabelling

        if labels and len(labels[xi]):

            l = labels[xi]

            v = torch.zeros((len(l), nc + 5), device=x.device)

            v[:, :4] = l[:, 1:5]  # box

            v[:, 4] = 1.0  # conf

            v[range(len(l)), l[:, 0].long() + 5] = 1.0  # cls

            x = torch.cat((x, v), 0)

        # If none remain process next image

        if not x.shape[0]:

            continue

        # Compute conf

        x[:, 5:] \*= x[:, 4:5]  # conf = obj\_conf \* cls\_conf

        # Box (center x, center y, width, height) to (x1, y1, x2, y2)

        box = xywh2xyxy(x[:, :4])

        # Detections matrix nx6 (xyxy, conf, cls)

        if multi\_label:

            i, j = (x[:, 5:] > conf\_thres).nonzero(as\_tuple=False).T

            x = torch.cat((box[i], x[i, j + 5, None], j[:, None].float()), 1)

        else:  # best class only

            conf, j = x[:, 5:].max(1, keepdim=True)

            x = torch.cat((box, conf, j.float()), 1)[conf.view(-1) > conf\_thres]

        # Filter by class

        if classes is not None:

            x = x[(x[:, 5:6] == torch.tensor(classes, device=x.device)).any(1)]

        # Apply finite constraint

        # if not torch.isfinite(x).all():

        #     x = x[torch.isfinite(x).all(1)]

        # Check shape

        n = x.shape[0]  # number of boxes

        if not n:  # no boxes

            continue

        elif n > max\_nms:  # excess boxes

            x = x[x[:, 4].argsort(descending=True)[:max\_nms]]  # sort by confidence

        # Batched NMS

        c = x[:, 5:6] \* (0 if agnostic else max\_wh)  # classes

        boxes, scores = x[:, :4] + c, x[:, 4]  # boxes (offset by class), scores

        i = torchvision.ops.nms(boxes, scores, iou\_thres)  # NMS

        if i.shape[0] > max\_det:  # limit detections

            i = i[:max\_det]

        if merge and (1 < n < 3E3):  # Merge NMS (boxes merged using weighted mean)

            # update boxes as boxes(i,4) = weights(i,n) \* boxes(n,4)

            iou = box\_iou(boxes[i], boxes) > iou\_thres  # iou matrix

            weights = iou \* scores[None]  # box weights

            x[i, :4] = torch.mm(weights, x[:, :4]).float() / weights.sum(1, keepdim=True)  # merged boxes

            if redundant:

                i = i[iou.sum(1) > 1]  # require redundancy

        output[xi] = x[i]

        if (time.time() - t) > time\_limit:

            print(f'WARNING: NMS time limit {time\_limit}s exceeded')

            break  # time limit exceeded

    return output

def strip\_optimizer(f='best.pt', s=''):  # from utils.general import \*; strip\_optimizer()

    # Strip optimizer from 'f' to finalize training, optionally save as 's'

    x = torch.load(f, map\_location=torch.device('cpu'))

    if x.get('ema'):

        x['model'] = x['ema']  # replace model with ema

    for k in 'optimizer', 'training\_results', 'wandb\_id', 'ema', 'updates':  # keys

        x[k] = None

    x['epoch'] = -1

    x['model'].half()  # to FP16

    for p in x['model'].parameters():

        p.requires\_grad = False

    torch.save(x, s or f)

    mb = os.path.getsize(s or f) / 1E6  # filesize

    print(f"Optimizer stripped from {f},{(' saved as %s,' % s) if s else ''} {mb:.1f}MB")

def print\_mutation(hyp, results, yaml\_file='hyp\_evolved.yaml', bucket=''):

    # Print mutation results to evolve.txt (for use with train.py --evolve)

    a = '%10s' \* len(hyp) % tuple(hyp.keys())  # hyperparam keys

    b = '%10.3g' \* len(hyp) % tuple(hyp.values())  # hyperparam values

    c = '%10.4g' \* len(results) % results  # results (P, R, mAP@0.5, mAP@0.5:0.95, val\_losses x 3)

    print('\n%s\n%s\nEvolved fitness: %s\n' % (a, b, c))

    if bucket:

        url = 'gs://%s/evolve.txt' % bucket

        if gsutil\_getsize(url) > (os.path.getsize('evolve.txt') if os.path.exists('evolve.txt') else 0):

            os.system('gsutil cp %s .' % url)  # download evolve.txt if larger than local

    with open('evolve.txt', 'a') as f:  # append result

        f.write(c + b + '\n')

    x = np.unique(np.loadtxt('evolve.txt', ndmin=2), axis=0)  # load unique rows

    x = x[np.argsort(-fitness(x))]  # sort

    np.savetxt('evolve.txt', x, '%10.3g')  # save sort by fitness

    # Save yaml

    for i, k in enumerate(hyp.keys()):

        hyp[k] = float(x[0, i + 7])

    with open(yaml\_file, 'w') as f:

        results = tuple(x[0, :7])

        c = '%10.4g' \* len(results) % results  # results (P, R, mAP@0.5, mAP@0.5:0.95, val\_losses x 3)

        f.write('# Hyperparameter Evolution Results\n# Generations: %g\n# Metrics: ' % len(x) + c + '\n\n')

        yaml.safe\_dump(hyp, f, sort\_keys=False)

    if bucket:

        os.system('gsutil cp evolve.txt %s gs://%s' % (yaml\_file, bucket))  # upload

def apply\_classifier(x, model, img, im0):

    # Apply a second stage classifier to yolo outputs

    im0 = [im0] if isinstance(im0, np.ndarray) else im0

    for i, d in enumerate(x):  # per image

        if d is not None and len(d):

            d = d.clone()

            # Reshape and pad cutouts

            b = xyxy2xywh(d[:, :4])  # boxes

            b[:, 2:] = b[:, 2:].max(1)[0].unsqueeze(1)  # rectangle to square

            b[:, 2:] = b[:, 2:] \* 1.3 + 30  # pad

            d[:, :4] = xywh2xyxy(b).long()

            # Rescale boxes from img\_size to im0 size

            scale\_coords(img.shape[2:], d[:, :4], im0[i].shape)

            # Classes

            pred\_cls1 = d[:, 5].long()

            ims = []

            for j, a in enumerate(d):  # per item

                cutout = im0[i][int(a[1]):int(a[3]), int(a[0]):int(a[2])]

                im = cv2.resize(cutout, (224, 224))  # BGR

                # cv2.imwrite('test%i.jpg' % j, cutout)

                im = im[:, :, ::-1].transpose(2, 0, 1)  # BGR to RGB, to 3x416x416

                im = np.ascontiguousarray(im, dtype=np.float32)  # uint8 to float32

                im /= 255.0  # 0 - 255 to 0.0 - 1.0

                ims.append(im)

            pred\_cls2 = model(torch.Tensor(ims).to(d.device)).argmax(1)  # classifier prediction

            x[i] = x[i][pred\_cls1 == pred\_cls2]  # retain matching class detections

    return x

def save\_one\_box(xyxy, im, file='image.jpg', gain=1.02, pad=10, square=False, BGR=False):

    # Save an image crop as {file} with crop size multiplied by {gain} and padded by {pad} pixels

    xyxy = torch.tensor(xyxy).view(-1, 4)

    b = xyxy2xywh(xyxy)  # boxes

    if square:

        b[:, 2:] = b[:, 2:].max(1)[0].unsqueeze(1)  # attempt rectangle to square

    b[:, 2:] = b[:, 2:] \* gain + pad  # box wh \* gain + pad

    xyxy = xywh2xyxy(b).long()

    clip\_coords(xyxy, im.shape)

    crop = im[int(xyxy[0, 1]):int(xyxy[0, 3]), int(xyxy[0, 0]):int(xyxy[0, 2])]

    cv2.imwrite(str(increment\_path(file, mkdir=True).with\_suffix('.jpg')), crop if BGR else crop[..., ::-1])

def increment\_path(path, exist\_ok=False, sep='', mkdir=False):

    # Increment file or directory path, i.e. runs/exp --> runs/exp{sep}2, runs/exp{sep}3, ... etc.

    path = Path(path)  # os-agnostic

    if path.exists() and not exist\_ok:

        suffix = path.suffix

        path = path.with\_suffix('')

        dirs = glob.glob(f"{path}{sep}\*")  # similar paths

        matches = [re.search(rf"%s{sep}(\d+)" % path.stem, d) for d in dirs]

        i = [int(m.groups()[0]) for m in matches if m]  # indices

        n = max(i) + 1 if i else 2  # increment number

        path = Path(f"{path}{sep}{n}{suffix}")  # update path

    dir = path if path.suffix == '' else path.parent  # directory

    if not dir.exists() and mkdir:

        dir.mkdir(parents=True, exist\_ok=True)  # make directory

    return path

yolo/utils.datasets.py dataloaders and functions for dataset augmentation

# Dataset utils and dataloaders

import glob

import logging

import math

import os

import random

import shutil

import time

from itertools import repeat

from multiprocessing.pool import ThreadPool

from pathlib import Path

from threading import Thread

import cv2

import numpy as np

import torch

import torch.nn.functional as F

from PIL import Image, ExifTags

from torch.utils.data import Dataset

from tqdm import tqdm

from utils.general import check\_requirements, xyxy2xywh, xywh2xyxy, xywhn2xyxy, xyn2xy, segment2box, segments2boxes, \

    resample\_segments, clean\_str

from utils.torch\_utils import torch\_distributed\_zero\_first

# Parameters

help\_url = 'https://github.com/ultralytics/yolov5/wiki/Train-Custom-Data'

img\_formats = ['bmp', 'jpg', 'jpeg', 'png', 'tif', 'tiff', 'dng', 'webp', 'mpo']  # acceptable image suffixes

vid\_formats = ['mov', 'avi', 'mp4', 'mpg', 'mpeg', 'm4v', 'wmv', 'mkv']  # acceptable video suffixes

logger = logging.getLogger(\_\_name\_\_)

# Get orientation exif tag

for orientation in ExifTags.TAGS.keys():

    if ExifTags.TAGS[orientation] == 'Orientation':

        break

def get\_hash(files):

    # Returns a single hash value of a list of files

    return sum(os.path.getsize(f) for f in files if os.path.isfile(f))

def exif\_size(img):

    # Returns exif-corrected PIL size

    s = img.size  # (width, height)

    try:

        rotation = dict(img.\_getexif().items())[orientation]

        if rotation == 6:  # rotation 270

            s = (s[1], s[0])

        elif rotation == 8:  # rotation 90

            s = (s[1], s[0])

    except:

        pass

    return s

def create\_dataloader(path, imgsz, batch\_size, stride, opt, hyp=None, augment=False, cache=False, pad=0.0, rect=False,

                      rank=-1, world\_size=1, workers=8, image\_weights=False, quad=False, prefix=''):

    # Make sure only the first process in DDP process the dataset first, and the following others can use the cache

    with torch\_distributed\_zero\_first(rank):

        dataset = LoadImagesAndLabels(path, imgsz, batch\_size,

                                      augment=augment,  # augment images

                                      hyp=hyp,  # augmentation hyperparameters

                                      rect=rect,  # rectangular training

                                      cache\_images=cache,

                                      single\_cls=opt.single\_cls,

                                      stride=int(stride),

                                      pad=pad,

                                      image\_weights=image\_weights,

                                      prefix=prefix)

    batch\_size = min(batch\_size, len(dataset))

    nw = min([os.cpu\_count() // world\_size, batch\_size if batch\_size > 1 else 0, workers])  # number of workers

    sampler = torch.utils.data.distributed.DistributedSampler(dataset) if rank != -1 else None

    loader = torch.utils.data.DataLoader if image\_weights else InfiniteDataLoader

    # Use torch.utils.data.DataLoader() if dataset.properties will update during training else InfiniteDataLoader()

    dataloader = loader(dataset,

                        batch\_size=batch\_size,

                        num\_workers=nw,

                        sampler=sampler,

                        pin\_memory=True,

                        collate\_fn=LoadImagesAndLabels.collate\_fn4 if quad else LoadImagesAndLabels.collate\_fn)

    return dataloader, dataset

class InfiniteDataLoader(torch.utils.data.dataloader.DataLoader):

    """ Dataloader that reuses workers

    Uses same syntax as vanilla DataLoader

    """

    def \_\_init\_\_(self, \*args, \*\*kwargs):

        super().\_\_init\_\_(\*args, \*\*kwargs)

        object.\_\_setattr\_\_(self, 'batch\_sampler', \_RepeatSampler(self.batch\_sampler))

        self.iterator = super().\_\_iter\_\_()

    def \_\_len\_\_(self):

        return len(self.batch\_sampler.sampler)

    def \_\_iter\_\_(self):

        for i in range(len(self)):

            yield next(self.iterator)

class \_RepeatSampler(object):

    """ Sampler that repeats forever

    Args:

        sampler (Sampler)

    """

    def \_\_init\_\_(self, sampler):

        self.sampler = sampler

    def \_\_iter\_\_(self):

        while True:

            yield from iter(self.sampler)

class LoadImages:  # for inference

    def \_\_init\_\_(self, path, img\_size=640, stride=32):

        p = str(Path(path).absolute())  # os-agnostic absolute path

        if '\*' in p:

            files = sorted(glob.glob(p, recursive=True))  # glob

        elif os.path.isdir(p):

            files = sorted(glob.glob(os.path.join(p, '\*.\*')))  # dir

        elif os.path.isfile(p):

            files = [p]  # files

        else:

            raise Exception(f'ERROR: {p} does not exist')

        images = [x for x in files if x.split('.')[-1].lower() in img\_formats]

        videos = [x for x in files if x.split('.')[-1].lower() in vid\_formats]

        ni, nv = len(images), len(videos)

        self.img\_size = img\_size

        self.stride = stride

        self.files = images + videos

        self.nf = ni + nv  # number of files

        self.video\_flag = [False] \* ni + [True] \* nv

        self.mode = 'image'

        if any(videos):

            self.new\_video(videos[0])  # new video

        else:

            self.cap = None

        assert self.nf > 0, f'No images or videos found in {p}. ' \

                            f'Supported formats are:\nimages: {img\_formats}\nvideos: {vid\_formats}'

    def \_\_iter\_\_(self):

        self.count = 0

        return self

    def \_\_next\_\_(self):

        if self.count == self.nf:

            raise StopIteration

        path = self.files[self.count]

        if self.video\_flag[self.count]:

            # Read video

            self.mode = 'video'

            ret\_val, img0 = self.cap.read()

            if not ret\_val:

                self.count += 1

                self.cap.release()

                if self.count == self.nf:  # last video

                    raise StopIteration

                else:

                    path = self.files[self.count]

                    self.new\_video(path)

                    ret\_val, img0 = self.cap.read()

            self.frame += 1

            print(f'video {self.count + 1}/{self.nf} ({self.frame}/{self.nframes}) {path}: ', end='')

        else:

            # Read image

            self.count += 1

            img0 = cv2.imread(path)  # BGR

            assert img0 is not None, 'Image Not Found ' + path

            print(f'image {self.count}/{self.nf} {path}: ', end='')

        # Padded resize

        img = letterbox(img0, self.img\_size, stride=self.stride)[0]

        # Convert

        img = img[:, :, ::-1].transpose(2, 0, 1)  # BGR to RGB, to 3x416x416

        img = np.ascontiguousarray(img)

        return path, img, img0, self.cap

    def new\_video(self, path):

        self.frame = 0

        self.cap = cv2.VideoCapture(path)

        self.nframes = int(self.cap.get(cv2.CAP\_PROP\_FRAME\_COUNT))

    def \_\_len\_\_(self):

        return self.nf  # number of files

class LoadWebcam:  # for inference

    def \_\_init\_\_(self, pipe='0', img\_size=640, stride=32):

        self.img\_size = img\_size

        self.stride = stride

        if pipe.isnumeric():

            pipe = eval(pipe)  # local camera

        # pipe = 'rtsp://192.168.1.64/1'  # IP camera

        # pipe = 'rtsp://username:password@192.168.1.64/1'  # IP camera with login

        # pipe = 'http://wmccpinetop.axiscam.net/mjpg/video.mjpg'  # IP golf camera

        self.pipe = pipe

        self.cap = cv2.VideoCapture(pipe)  # video capture object

        self.cap.set(cv2.CAP\_PROP\_BUFFERSIZE, 3)  # set buffer size

    def \_\_iter\_\_(self):

        self.count = -1

        return self

    def \_\_next\_\_(self):

        self.count += 1

        if cv2.waitKey(1) == ord('q'):  # q to quit

            self.cap.release()

            cv2.destroyAllWindows()

            raise StopIteration

        # Read frame

        if self.pipe == 0:  # local camera

            ret\_val, img0 = self.cap.read()

            img0 = cv2.flip(img0, 1)  # flip left-right

        else:  # IP camera

            n = 0

            while True:

                n += 1

                self.cap.grab()

                if n % 30 == 0:  # skip frames

                    ret\_val, img0 = self.cap.retrieve()

                    if ret\_val:

                        break

        # Print

        assert ret\_val, f'Camera Error {self.pipe}'

        img\_path = 'webcam.jpg'

        print(f'webcam {self.count}: ', end='')

        # Padded resize

        img = letterbox(img0, self.img\_size, stride=self.stride)[0]

        # Convert

        img = img[:, :, ::-1].transpose(2, 0, 1)  # BGR to RGB, to 3x416x416

        img = np.ascontiguousarray(img)

        return img\_path, img, img0, None

    def \_\_len\_\_(self):

        return 0

class LoadStreams:  # multiple IP or RTSP cameras

    def \_\_init\_\_(self, sources='streams.txt', img\_size=640, stride=32):

        self.mode = 'stream'

        self.img\_size = img\_size

        self.stride = stride

        if os.path.isfile(sources):

            with open(sources, 'r') as f:

                sources = [x.strip() for x in f.read().strip().splitlines() if len(x.strip())]

        else:

            sources = [sources]

        n = len(sources)

        self.imgs = [None] \* n

        self.sources = [clean\_str(x) for x in sources]  # clean source names for later

        for i, s in enumerate(sources):  # index, source

            # Start thread to read frames from video stream

            print(f'{i + 1}/{n}: {s}... ', end='')

            if 'youtube.com/' in s or 'youtu.be/' in s:  # if source is YouTube video

                check\_requirements(('pafy', 'youtube\_dl'))

                import pafy

                s = pafy.new(s).getbest(preftype="mp4").url  # YouTube URL

            s = eval(s) if s.isnumeric() else s  # i.e. s = '0' local webcam

            cap = cv2.VideoCapture(s)

            assert cap.isOpened(), f'Failed to open {s}'

            w = int(cap.get(cv2.CAP\_PROP\_FRAME\_WIDTH))

            h = int(cap.get(cv2.CAP\_PROP\_FRAME\_HEIGHT))

            self.fps = cap.get(cv2.CAP\_PROP\_FPS) % 100

            \_, self.imgs[i] = cap.read()  # guarantee first frame

            thread = Thread(target=self.update, args=([i, cap]), daemon=True)

            print(f' success ({w}x{h} at {self.fps:.2f} FPS).')

            thread.start()

        print('')  # newline

        # check for common shapes

        s = np.stack([letterbox(x, self.img\_size, stride=self.stride)[0].shape for x in self.imgs], 0)  # shapes

        self.rect = np.unique(s, axis=0).shape[0] == 1  # rect inference if all shapes equal

        if not self.rect:

            print('WARNING: Different stream shapes detected. For optimal performance supply similarly-shaped streams.')

    def update(self, index, cap):

        # Read next stream frame in a daemon thread

        n = 0

        while cap.isOpened():

            n += 1

            # \_, self.imgs[index] = cap.read()

            cap.grab()

            if n == 4:  # read every 4th frame

                success, im = cap.retrieve()

                self.imgs[index] = im if success else self.imgs[index] \* 0

                n = 0

            time.sleep(1 / self.fps)  # wait time

    def \_\_iter\_\_(self):

        self.count = -1

        return self

    def \_\_next\_\_(self):

        self.count += 1

        img0 = self.imgs.copy()

        if cv2.waitKey(1) == ord('q'):  # q to quit

            cv2.destroyAllWindows()

            raise StopIteration

        # Letterbox

        img = [letterbox(x, self.img\_size, auto=self.rect, stride=self.stride)[0] for x in img0]

        # Stack

        img = np.stack(img, 0)

        # Convert

        img = img[:, :, :, ::-1].transpose(0, 3, 1, 2)  # BGR to RGB, to bsx3x416x416

        img = np.ascontiguousarray(img)

        return self.sources, img, img0, None

    def \_\_len\_\_(self):

        return 0  # 1E12 frames = 32 streams at 30 FPS for 30 years

def img2label\_paths(img\_paths):

    # Define label paths as a function of image paths

    sa, sb = os.sep + 'images' + os.sep, os.sep + 'labels' + os.sep  # /images/, /labels/ substrings

    return ['txt'.join(x.replace(sa, sb, 1).rsplit(x.split('.')[-1], 1)) for x in img\_paths]

class LoadImagesAndLabels(Dataset):  # for training/testing

    def \_\_init\_\_(self, path, img\_size=640, batch\_size=16, augment=False, hyp=None, rect=False, image\_weights=False,

                 cache\_images=False, single\_cls=False, stride=32, pad=0.0, prefix=''):

        self.img\_size = img\_size

        self.augment = augment

        self.hyp = hyp

        self.image\_weights = image\_weights

        self.rect = False if image\_weights else rect

        self.mosaic = self.augment and not self.rect  # load 4 images at a time into a mosaic (only during training)

        self.mosaic\_border = [-img\_size // 2, -img\_size // 2]

        self.stride = stride

        self.path = path

        try:

            f = []  # image files

            for p in path if isinstance(path, list) else [path]:

                p = Path(p)  # os-agnostic

                if p.is\_dir():  # dir

                    f += glob.glob(str(p / '\*\*' / '\*.\*'), recursive=True)

                    # f = list(p.rglob('\*\*/\*.\*'))  # pathlib

                elif p.is\_file():  # file

                    with open(p, 'r') as t:

                        t = t.read().strip().splitlines()

                        parent = str(p.parent) + os.sep

                        f += [x.replace('./', parent) if x.startswith('./') else x for x in t]  # local to global path

                        # f += [p.parent / x.lstrip(os.sep) for x in t]  # local to global path (pathlib)

                else:

                    raise Exception(f'{prefix}{p} does not exist')

            self.img\_files = sorted([x.replace('/', os.sep) for x in f if x.split('.')[-1].lower() in img\_formats])

            # self.img\_files = sorted([x for x in f if x.suffix[1:].lower() in img\_formats])  # pathlib

            assert self.img\_files, f'{prefix}No images found'

        except Exception as e:

            raise Exception(f'{prefix}Error loading data from {path}: {e}\nSee {help\_url}')

        # Check cache

        self.label\_files = img2label\_paths(self.img\_files)  # labels

        cache\_path = (p if p.is\_file() else Path(self.label\_files[0]).parent).with\_suffix('.cache')  # cached labels

        if cache\_path.is\_file():

            cache, exists = torch.load(cache\_path), True  # load

            if cache['hash'] != get\_hash(self.label\_files + self.img\_files) or 'version' not in cache:  # changed

                cache, exists = self.cache\_labels(cache\_path, prefix), False  # re-cache

        else:

            cache, exists = self.cache\_labels(cache\_path, prefix), False  # cache

        # Display cache

        nf, nm, ne, nc, n = cache.pop('results')  # found, missing, empty, corrupted, total

        if exists:

            d = f"Scanning '{cache\_path}' images and labels... {nf} found, {nm} missing, {ne} empty, {nc} corrupted"

            tqdm(None, desc=prefix + d, total=n, initial=n)  # display cache results

        assert nf > 0 or not augment, f'{prefix}No labels in {cache\_path}. Can not train without labels. See {help\_url}'

        # Read cache

        cache.pop('hash')  # remove hash

        cache.pop('version')  # remove version

        labels, shapes, self.segments = zip(\*cache.values())

        self.labels = list(labels)

        self.shapes = np.array(shapes, dtype=np.float64)

        self.img\_files = list(cache.keys())  # update

        self.label\_files = img2label\_paths(cache.keys())  # update

        if single\_cls:

            for x in self.labels:

                x[:, 0] = 0

        n = len(shapes)  # number of images

        bi = np.floor(np.arange(n) / batch\_size).astype(np.int)  # batch index

        nb = bi[-1] + 1  # number of batches

        self.batch = bi  # batch index of image

        self.n = n

        self.indices = range(n)

        # Rectangular Training

        if self.rect:

            # Sort by aspect ratio

            s = self.shapes  # wh

            ar = s[:, 1] / s[:, 0]  # aspect ratio

            irect = ar.argsort()

            self.img\_files = [self.img\_files[i] for i in irect]

            self.label\_files = [self.label\_files[i] for i in irect]

            self.labels = [self.labels[i] for i in irect]

            self.shapes = s[irect]  # wh

            ar = ar[irect]

            # Set training image shapes

            shapes = [[1, 1]] \* nb

            for i in range(nb):

                ari = ar[bi == i]

                mini, maxi = ari.min(), ari.max()

                if maxi < 1:

                    shapes[i] = [maxi, 1]

                elif mini > 1:

                    shapes[i] = [1, 1 / mini]

            self.batch\_shapes = np.ceil(np.array(shapes) \* img\_size / stride + pad).astype(np.int) \* stride

        # Cache images into memory for faster training (WARNING: large datasets may exceed system RAM)

        self.imgs = [None] \* n

        if cache\_images:

            gb = 0  # Gigabytes of cached images

            self.img\_hw0, self.img\_hw = [None] \* n, [None] \* n

            results = ThreadPool(8).imap(lambda x: load\_image(\*x), zip(repeat(self), range(n)))  # 8 threads

            pbar = tqdm(enumerate(results), total=n)

            for i, x in pbar:

                self.imgs[i], self.img\_hw0[i], self.img\_hw[i] = x  # img, hw\_original, hw\_resized = load\_image(self, i)

                gb += self.imgs[i].nbytes

                pbar.desc = f'{prefix}Caching images ({gb / 1E9:.1f}GB)'

            pbar.close()

    def cache\_labels(self, path=Path('./labels.cache'), prefix=''):

        # Cache dataset labels, check images and read shapes

        x = {}  # dict

        nm, nf, ne, nc = 0, 0, 0, 0  # number missing, found, empty, duplicate

        pbar = tqdm(zip(self.img\_files, self.label\_files), desc='Scanning images', total=len(self.img\_files))

        for i, (im\_file, lb\_file) in enumerate(pbar):

            try:

                # verify images

                im = Image.open(im\_file)

                im.verify()  # PIL verify

                shape = exif\_size(im)  # image size

                segments = []  # instance segments

                assert (shape[0] > 9) & (shape[1] > 9), f'image size {shape} <10 pixels'

                assert im.format.lower() in img\_formats, f'invalid image format {im.format}'

                # verify labels

                if os.path.isfile(lb\_file):

                    nf += 1  # label found

                    with open(lb\_file, 'r') as f:

                        l = [x.split() for x in f.read().strip().splitlines()]

                        if any([len(x) > 8 for x in l]):  # is segment

                            classes = np.array([x[0] for x in l], dtype=np.float32)

                            segments = [np.array(x[1:], dtype=np.float32).reshape(-1, 2) for x in l]  # (cls, xy1...)

                            l = np.concatenate((classes.reshape(-1, 1), segments2boxes(segments)), 1)  # (cls, xywh)

                        l = np.array(l, dtype=np.float32)

                    if len(l):

                        assert l.shape[1] == 5, 'labels require 5 columns each'

                        assert (l >= 0).all(), 'negative labels'

                        assert (l[:, 1:] <= 1).all(), 'non-normalized or out of bounds coordinate labels'

                        assert np.unique(l, axis=0).shape[0] == l.shape[0], 'duplicate labels'

                    else:

                        ne += 1  # label empty

                        l = np.zeros((0, 5), dtype=np.float32)

                else:

                    nm += 1  # label missing

                    l = np.zeros((0, 5), dtype=np.float32)

                x[im\_file] = [l, shape, segments]

            except Exception as e:

                nc += 1

                logging.info(f'{prefix}WARNING: Ignoring corrupted image and/or label {im\_file}: {e}')

            pbar.desc = f"{prefix}Scanning '{path.parent / path.stem}' images and labels... " \

                        f"{nf} found, {nm} missing, {ne} empty, {nc} corrupted"

        pbar.close()

        if nf == 0:

            logging.info(f'{prefix}WARNING: No labels found in {path}. See {help\_url}')

        x['hash'] = get\_hash(self.label\_files + self.img\_files)

        x['results'] = nf, nm, ne, nc, i + 1

        x['version'] = 0.1  # cache version

        try:

            torch.save(x, path)  # save for next time

            logging.info(f'{prefix}New cache created: {path}')

        except Exception as e:

            logging.info(f'{prefix}WARNING: Cache directory {path.parent} is not writeable: {e}')  # path not writeable

        return x

    def \_\_len\_\_(self):

        return len(self.img\_files)

    # def \_\_iter\_\_(self):

    #     self.count = -1

    #     print('ran dataset iter')

    #     #self.shuffled\_vector = np.random.permutation(self.nF) if self.augment else np.arange(self.nF)

    #     return self

    def \_\_getitem\_\_(self, index):

        index = self.indices[index]  # linear, shuffled, or image\_weights

        hyp = self.hyp

        mosaic = self.mosaic and random.random() < hyp['mosaic']

        if mosaic:

            # Load mosaic

            img, labels = load\_mosaic(self, index)

            shapes = None

            # MixUp https://arxiv.org/pdf/1710.09412.pdf

            if random.random() < hyp['mixup']:

                img2, labels2 = load\_mosaic(self, random.randint(0, self.n - 1))

                r = np.random.beta(8.0, 8.0)  # mixup ratio, alpha=beta=8.0

                img = (img \* r + img2 \* (1 - r)).astype(np.uint8)

                labels = np.concatenate((labels, labels2), 0)

        else:

            # Load image

            img, (h0, w0), (h, w) = load\_image(self, index)

            # Letterbox

            shape = self.batch\_shapes[self.batch[index]] if self.rect else self.img\_size  # final letterboxed shape

            img, ratio, pad = letterbox(img, shape, auto=False, scaleup=self.augment)

            shapes = (h0, w0), ((h / h0, w / w0), pad)  # for COCO mAP rescaling

            labels = self.labels[index].copy()

            if labels.size:  # normalized xywh to pixel xyxy format

                labels[:, 1:] = xywhn2xyxy(labels[:, 1:], ratio[0] \* w, ratio[1] \* h, padw=pad[0], padh=pad[1])

        if self.augment:

            # Augment imagespace

            if not mosaic:

                img, labels = random\_perspective(img, labels,

                                                 degrees=hyp['degrees'],

                                                 translate=hyp['translate'],

                                                 scale=hyp['scale'],

                                                 shear=hyp['shear'],

                                                 perspective=hyp['perspective'])

            # Augment colorspace

            augment\_hsv(img, hgain=hyp['hsv\_h'], sgain=hyp['hsv\_s'], vgain=hyp['hsv\_v'])

            # Apply cutouts

            # if random.random() < 0.9:

            #     labels = cutout(img, labels)

        nL = len(labels)  # number of labels

        if nL:

            labels[:, 1:5] = xyxy2xywh(labels[:, 1:5])  # convert xyxy to xywh

            labels[:, [2, 4]] /= img.shape[0]  # normalized height 0-1

            labels[:, [1, 3]] /= img.shape[1]  # normalized width 0-1

        if self.augment:

            # flip up-down

            if random.random() < hyp['flipud']:

                img = np.flipud(img)

                if nL:

                    labels[:, 2] = 1 - labels[:, 2]

            # flip left-right

            if random.random() < hyp['fliplr']:

                img = np.fliplr(img)

                if nL:

                    labels[:, 1] = 1 - labels[:, 1]

        labels\_out = torch.zeros((nL, 6))

        if nL:

            labels\_out[:, 1:] = torch.from\_numpy(labels)

        # Convert

        img = img[:, :, ::-1].transpose(2, 0, 1)  # BGR to RGB, to 3x416x416

        img = np.ascontiguousarray(img)

        return torch.from\_numpy(img), labels\_out, self.img\_files[index], shapes

    @staticmethod

    def collate\_fn(batch):

        img, label, path, shapes = zip(\*batch)  # transposed

        for i, l in enumerate(label):

            l[:, 0] = i  # add target image index for build\_targets()

        return torch.stack(img, 0), torch.cat(label, 0), path, shapes

    @staticmethod

    def collate\_fn4(batch):

        img, label, path, shapes = zip(\*batch)  # transposed

        n = len(shapes) // 4

        img4, label4, path4, shapes4 = [], [], path[:n], shapes[:n]

        ho = torch.tensor([[0., 0, 0, 1, 0, 0]])

        wo = torch.tensor([[0., 0, 1, 0, 0, 0]])

        s = torch.tensor([[1, 1, .5, .5, .5, .5]])  # scale

        for i in range(n):  # zidane torch.zeros(16,3,720,1280)  # BCHW

            i \*= 4

            if random.random() < 0.5:

                im = F.interpolate(img[i].unsqueeze(0).float(), scale\_factor=2., mode='bilinear', align\_corners=False)[

                    0].type(img[i].type())

                l = label[i]

            else:

                im = torch.cat((torch.cat((img[i], img[i + 1]), 1), torch.cat((img[i + 2], img[i + 3]), 1)), 2)

                l = torch.cat((label[i], label[i + 1] + ho, label[i + 2] + wo, label[i + 3] + ho + wo), 0) \* s

            img4.append(im)

            label4.append(l)

        for i, l in enumerate(label4):

            l[:, 0] = i  # add target image index for build\_targets()

        return torch.stack(img4, 0), torch.cat(label4, 0), path4, shapes4

# Ancillary functions --------------------------------------------------------------------------------------------------

def load\_image(self, index):

    # loads 1 image from dataset, returns img, original hw, resized hw

    img = self.imgs[index]

    if img is None:  # not cached

        path = self.img\_files[index]

        img = cv2.imread(path)  # BGR

        assert img is not None, 'Image Not Found ' + path

        h0, w0 = img.shape[:2]  # orig hw

        r = self.img\_size / max(h0, w0)  # ratio

        if r != 1:  # if sizes are not equal

            img = cv2.resize(img, (int(w0 \* r), int(h0 \* r)),

                             interpolation=cv2.INTER\_AREA if r < 1 and not self.augment else cv2.INTER\_LINEAR)

        return img, (h0, w0), img.shape[:2]  # img, hw\_original, hw\_resized

    else:

        return self.imgs[index], self.img\_hw0[index], self.img\_hw[index]  # img, hw\_original, hw\_resized

def augment\_hsv(img, hgain=0.5, sgain=0.5, vgain=0.5):

    r = np.random.uniform(-1, 1, 3) \* [hgain, sgain, vgain] + 1  # random gains

    hue, sat, val = cv2.split(cv2.cvtColor(img, cv2.COLOR\_BGR2HSV))

    dtype = img.dtype  # uint8

    x = np.arange(0, 256, dtype=np.int16)

    lut\_hue = ((x \* r[0]) % 180).astype(dtype)

    lut\_sat = np.clip(x \* r[1], 0, 255).astype(dtype)

    lut\_val = np.clip(x \* r[2], 0, 255).astype(dtype)

    img\_hsv = cv2.merge((cv2.LUT(hue, lut\_hue), cv2.LUT(sat, lut\_sat), cv2.LUT(val, lut\_val))).astype(dtype)

    cv2.cvtColor(img\_hsv, cv2.COLOR\_HSV2BGR, dst=img)  # no return needed

def hist\_equalize(img, clahe=True, bgr=False):

    # Equalize histogram on BGR image 'img' with img.shape(n,m,3) and range 0-255

    yuv = cv2.cvtColor(img, cv2.COLOR\_BGR2YUV if bgr else cv2.COLOR\_RGB2YUV)

    if clahe:

        c = cv2.createCLAHE(clipLimit=2.0, tileGridSize=(8, 8))

        yuv[:, :, 0] = c.apply(yuv[:, :, 0])

    else:

        yuv[:, :, 0] = cv2.equalizeHist(yuv[:, :, 0])  # equalize Y channel histogram

    return cv2.cvtColor(yuv, cv2.COLOR\_YUV2BGR if bgr else cv2.COLOR\_YUV2RGB)  # convert YUV image to RGB

def load\_mosaic(self, index):

    # loads images in a 4-mosaic

    labels4, segments4 = [], []

    s = self.img\_size

    yc, xc = [int(random.uniform(-x, 2 \* s + x)) for x in self.mosaic\_border]  # mosaic center x, y

    indices = [index] + random.choices(self.indices, k=3)  # 3 additional image indices

    for i, index in enumerate(indices):

        # Load image

        img, \_, (h, w) = load\_image(self, index)

        # place img in img4

        if i == 0:  # top left

            img4 = np.full((s \* 2, s \* 2, img.shape[2]), 114, dtype=np.uint8)  # base image with 4 tiles

            x1a, y1a, x2a, y2a = max(xc - w, 0), max(yc - h, 0), xc, yc  # xmin, ymin, xmax, ymax (large image)

            x1b, y1b, x2b, y2b = w - (x2a - x1a), h - (y2a - y1a), w, h  # xmin, ymin, xmax, ymax (small image)

        elif i == 1:  # top right

            x1a, y1a, x2a, y2a = xc, max(yc - h, 0), min(xc + w, s \* 2), yc

            x1b, y1b, x2b, y2b = 0, h - (y2a - y1a), min(w, x2a - x1a), h

        elif i == 2:  # bottom left

            x1a, y1a, x2a, y2a = max(xc - w, 0), yc, xc, min(s \* 2, yc + h)

            x1b, y1b, x2b, y2b = w - (x2a - x1a), 0, w, min(y2a - y1a, h)

        elif i == 3:  # bottom right

            x1a, y1a, x2a, y2a = xc, yc, min(xc + w, s \* 2), min(s \* 2, yc + h)

            x1b, y1b, x2b, y2b = 0, 0, min(w, x2a - x1a), min(y2a - y1a, h)

        img4[y1a:y2a, x1a:x2a] = img[y1b:y2b, x1b:x2b]  # img4[ymin:ymax, xmin:xmax]

        padw = x1a - x1b

        padh = y1a - y1b

        # Labels

        labels, segments = self.labels[index].copy(), self.segments[index].copy()

        if labels.size:

            labels[:, 1:] = xywhn2xyxy(labels[:, 1:], w, h, padw, padh)  # normalized xywh to pixel xyxy format

            segments = [xyn2xy(x, w, h, padw, padh) for x in segments]

        labels4.append(labels)

        segments4.extend(segments)

    # Concat/clip labels

    labels4 = np.concatenate(labels4, 0)

    for x in (labels4[:, 1:], \*segments4):

        np.clip(x, 0, 2 \* s, out=x)  # clip when using random\_perspective()

    # img4, labels4 = replicate(img4, labels4)  # replicate

    # Augment

    img4, labels4 = random\_perspective(img4, labels4, segments4,

                                       degrees=self.hyp['degrees'],

                                       translate=self.hyp['translate'],

                                       scale=self.hyp['scale'],

                                       shear=self.hyp['shear'],

                                       perspective=self.hyp['perspective'],

                                       border=self.mosaic\_border)  # border to remove

    return img4, labels4

def load\_mosaic9(self, index):

    # loads images in a 9-mosaic

    labels9, segments9 = [], []

    s = self.img\_size

    indices = [index] + random.choices(self.indices, k=8)  # 8 additional image indices

    for i, index in enumerate(indices):

        # Load image

        img, \_, (h, w) = load\_image(self, index)

        # place img in img9

        if i == 0:  # center

            img9 = np.full((s \* 3, s \* 3, img.shape[2]), 114, dtype=np.uint8)  # base image with 4 tiles

            h0, w0 = h, w

            c = s, s, s + w, s + h  # xmin, ymin, xmax, ymax (base) coordinates

        elif i == 1:  # top

            c = s, s - h, s + w, s

        elif i == 2:  # top right

            c = s + wp, s - h, s + wp + w, s

        elif i == 3:  # right

            c = s + w0, s, s + w0 + w, s + h

        elif i == 4:  # bottom right

            c = s + w0, s + hp, s + w0 + w, s + hp + h

        elif i == 5:  # bottom

            c = s + w0 - w, s + h0, s + w0, s + h0 + h

        elif i == 6:  # bottom left

            c = s + w0 - wp - w, s + h0, s + w0 - wp, s + h0 + h

        elif i == 7:  # left

            c = s - w, s + h0 - h, s, s + h0

        elif i == 8:  # top left

            c = s - w, s + h0 - hp - h, s, s + h0 - hp

        padx, pady = c[:2]

        x1, y1, x2, y2 = [max(x, 0) for x in c]  # allocate coords

        # Labels

        labels, segments = self.labels[index].copy(), self.segments[index].copy()

        if labels.size:

            labels[:, 1:] = xywhn2xyxy(labels[:, 1:], w, h, padx, pady)  # normalized xywh to pixel xyxy format

            segments = [xyn2xy(x, w, h, padx, pady) for x in segments]

        labels9.append(labels)

        segments9.extend(segments)

        # Image

        img9[y1:y2, x1:x2] = img[y1 - pady:, x1 - padx:]  # img9[ymin:ymax, xmin:xmax]

        hp, wp = h, w  # height, width previous

    # Offset

    yc, xc = [int(random.uniform(0, s)) for \_ in self.mosaic\_border]  # mosaic center x, y

    img9 = img9[yc:yc + 2 \* s, xc:xc + 2 \* s]

    # Concat/clip labels

    labels9 = np.concatenate(labels9, 0)

    labels9[:, [1, 3]] -= xc

    labels9[:, [2, 4]] -= yc

    c = np.array([xc, yc])  # centers

    segments9 = [x - c for x in segments9]

    for x in (labels9[:, 1:], \*segments9):

        np.clip(x, 0, 2 \* s, out=x)  # clip when using random\_perspective()

    # img9, labels9 = replicate(img9, labels9)  # replicate

    # Augment

    img9, labels9 = random\_perspective(img9, labels9, segments9,

                                       degrees=self.hyp['degrees'],

                                       translate=self.hyp['translate'],

                                       scale=self.hyp['scale'],

                                       shear=self.hyp['shear'],

                                       perspective=self.hyp['perspective'],

                                       border=self.mosaic\_border)  # border to remove

    return img9, labels9

def replicate(img, labels):

    # Replicate labels

    h, w = img.shape[:2]

    boxes = labels[:, 1:].astype(int)

    x1, y1, x2, y2 = boxes.T

    s = ((x2 - x1) + (y2 - y1)) / 2  # side length (pixels)

    for i in s.argsort()[:round(s.size \* 0.5)]:  # smallest indices

        x1b, y1b, x2b, y2b = boxes[i]

        bh, bw = y2b - y1b, x2b - x1b

        yc, xc = int(random.uniform(0, h - bh)), int(random.uniform(0, w - bw))  # offset x, y

        x1a, y1a, x2a, y2a = [xc, yc, xc + bw, yc + bh]

        img[y1a:y2a, x1a:x2a] = img[y1b:y2b, x1b:x2b]  # img4[ymin:ymax, xmin:xmax]

        labels = np.append(labels, [[labels[i, 0], x1a, y1a, x2a, y2a]], axis=0)

    return img, labels

def letterbox(img, new\_shape=(640, 640), color=(114, 114, 114), auto=True, scaleFill=False, scaleup=True, stride=32):

    # Resize and pad image while meeting stride-multiple constraints

    shape = img.shape[:2]  # current shape [height, width]

    if isinstance(new\_shape, int):

        new\_shape = (new\_shape, new\_shape)

    # Scale ratio (new / old)

    r = min(new\_shape[0] / shape[0], new\_shape[1] / shape[1])

    if not scaleup:  # only scale down, do not scale up (for better test mAP)

        r = min(r, 1.0)

    # Compute padding

    ratio = r, r  # width, height ratios

    new\_unpad = int(round(shape[1] \* r)), int(round(shape[0] \* r))

    dw, dh = new\_shape[1] - new\_unpad[0], new\_shape[0] - new\_unpad[1]  # wh padding

    if auto:  # minimum rectangle

        dw, dh = np.mod(dw, stride), np.mod(dh, stride)  # wh padding

    elif scaleFill:  # stretch

        dw, dh = 0.0, 0.0

        new\_unpad = (new\_shape[1], new\_shape[0])

        ratio = new\_shape[1] / shape[1], new\_shape[0] / shape[0]  # width, height ratios

    dw /= 2  # divide padding into 2 sides

    dh /= 2

    if shape[::-1] != new\_unpad:  # resize

        img = cv2.resize(img, new\_unpad, interpolation=cv2.INTER\_LINEAR)

    top, bottom = int(round(dh - 0.1)), int(round(dh + 0.1))

    left, right = int(round(dw - 0.1)), int(round(dw + 0.1))

    img = cv2.copyMakeBorder(img, top, bottom, left, right, cv2.BORDER\_CONSTANT, value=color)  # add border

    return img, ratio, (dw, dh)

def random\_perspective(img, targets=(), segments=(), degrees=10, translate=.1, scale=.1, shear=10, perspective=0.0,

                       border=(0, 0)):

    # torchvision.transforms.RandomAffine(degrees=(-10, 10), translate=(.1, .1), scale=(.9, 1.1), shear=(-10, 10))

    # targets = [cls, xyxy]

    height = img.shape[0] + border[0] \* 2  # shape(h,w,c)

    width = img.shape[1] + border[1] \* 2

    # Center

    C = np.eye(3)

    C[0, 2] = -img.shape[1] / 2  # x translation (pixels)

    C[1, 2] = -img.shape[0] / 2  # y translation (pixels)

    # Perspective

    P = np.eye(3)

    P[2, 0] = random.uniform(-perspective, perspective)  # x perspective (about y)

    P[2, 1] = random.uniform(-perspective, perspective)  # y perspective (about x)

    # Rotation and Scale

    R = np.eye(3)

    a = random.uniform(-degrees, degrees)

    # a += random.choice([-180, -90, 0, 90])  # add 90deg rotations to small rotations

    s = random.uniform(1 - scale, 1 + scale)

    # s = 2 \*\* random.uniform(-scale, scale)

    R[:2] = cv2.getRotationMatrix2D(angle=a, center=(0, 0), scale=s)

    # Shear

    S = np.eye(3)

    S[0, 1] = math.tan(random.uniform(-shear, shear) \* math.pi / 180)  # x shear (deg)

    S[1, 0] = math.tan(random.uniform(-shear, shear) \* math.pi / 180)  # y shear (deg)

    # Translation

    T = np.eye(3)

    T[0, 2] = random.uniform(0.5 - translate, 0.5 + translate) \* width  # x translation (pixels)

    T[1, 2] = random.uniform(0.5 - translate, 0.5 + translate) \* height  # y translation (pixels)

    # Combined rotation matrix

    M = T @ S @ R @ P @ C  # order of operations (right to left) is IMPORTANT

    if (border[0] != 0) or (border[1] != 0) or (M != np.eye(3)).any():  # image changed

        if perspective:

            img = cv2.warpPerspective(img, M, dsize=(width, height), borderValue=(114, 114, 114))

        else:  # affine

            img = cv2.warpAffine(img, M[:2], dsize=(width, height), borderValue=(114, 114, 114))

    # Visualize

    # import matplotlib.pyplot as plt

    # ax = plt.subplots(1, 2, figsize=(12, 6))[1].ravel()

    # ax[0].imshow(img[:, :, ::-1])  # base

    # ax[1].imshow(img2[:, :, ::-1])  # warped

    # Transform label coordinates

    n = len(targets)

    if n:

        use\_segments = any(x.any() for x in segments)

        new = np.zeros((n, 4))

        if use\_segments:  # warp segments

            segments = resample\_segments(segments)  # upsample

            for i, segment in enumerate(segments):

                xy = np.ones((len(segment), 3))

                xy[:, :2] = segment

                xy = xy @ M.T  # transform

                xy = xy[:, :2] / xy[:, 2:3] if perspective else xy[:, :2]  # perspective rescale or affine

                # clip

                new[i] = segment2box(xy, width, height)

        else:  # warp boxes

            xy = np.ones((n \* 4, 3))

            xy[:, :2] = targets[:, [1, 2, 3, 4, 1, 4, 3, 2]].reshape(n \* 4, 2)  # x1y1, x2y2, x1y2, x2y1

            xy = xy @ M.T  # transform

            xy = (xy[:, :2] / xy[:, 2:3] if perspective else xy[:, :2]).reshape(n, 8)  # perspective rescale or affine

            # create new boxes

            x = xy[:, [0, 2, 4, 6]]

            y = xy[:, [1, 3, 5, 7]]

            new = np.concatenate((x.min(1), y.min(1), x.max(1), y.max(1))).reshape(4, n).T

            # clip

            new[:, [0, 2]] = new[:, [0, 2]].clip(0, width)

            new[:, [1, 3]] = new[:, [1, 3]].clip(0, height)

        # filter candidates

        i = box\_candidates(box1=targets[:, 1:5].T \* s, box2=new.T, area\_thr=0.01 if use\_segments else 0.10)

        targets = targets[i]

        targets[:, 1:5] = new[i]

    return img, targets

def box\_candidates(box1, box2, wh\_thr=2, ar\_thr=20, area\_thr=0.1, eps=1e-16):  # box1(4,n), box2(4,n)

    # Compute candidate boxes: box1 before augment, box2 after augment, wh\_thr (pixels), aspect\_ratio\_thr, area\_ratio

    w1, h1 = box1[2] - box1[0], box1[3] - box1[1]

    w2, h2 = box2[2] - box2[0], box2[3] - box2[1]

    ar = np.maximum(w2 / (h2 + eps), h2 / (w2 + eps))  # aspect ratio

    return (w2 > wh\_thr) & (h2 > wh\_thr) & (w2 \* h2 / (w1 \* h1 + eps) > area\_thr) & (ar < ar\_thr)  # candidates

def cutout(image, labels):

    # Applies image cutout augmentation https://arxiv.org/abs/1708.04552

    h, w = image.shape[:2]

    def bbox\_ioa(box1, box2):

        # Returns the intersection over box2 area given box1, box2. box1 is 4, box2 is nx4. boxes are x1y1x2y2

        box2 = box2.transpose()

        # Get the coordinates of bounding boxes

        b1\_x1, b1\_y1, b1\_x2, b1\_y2 = box1[0], box1[1], box1[2], box1[3]

        b2\_x1, b2\_y1, b2\_x2, b2\_y2 = box2[0], box2[1], box2[2], box2[3]

        # Intersection area

        inter\_area = (np.minimum(b1\_x2, b2\_x2) - np.maximum(b1\_x1, b2\_x1)).clip(0) \* \

                     (np.minimum(b1\_y2, b2\_y2) - np.maximum(b1\_y1, b2\_y1)).clip(0)

        # box2 area

        box2\_area = (b2\_x2 - b2\_x1) \* (b2\_y2 - b2\_y1) + 1e-16

        # Intersection over box2 area

        return inter\_area / box2\_area

    # create random masks

    scales = [0.5] \* 1 + [0.25] \* 2 + [0.125] \* 4 + [0.0625] \* 8 + [0.03125] \* 16  # image size fraction

    for s in scales:

        mask\_h = random.randint(1, int(h \* s))

        mask\_w = random.randint(1, int(w \* s))

        # box

        xmin = max(0, random.randint(0, w) - mask\_w // 2)

        ymin = max(0, random.randint(0, h) - mask\_h // 2)

        xmax = min(w, xmin + mask\_w)

        ymax = min(h, ymin + mask\_h)

        # apply random color mask

        image[ymin:ymax, xmin:xmax] = [random.randint(64, 191) for \_ in range(3)]

        # return unobscured labels

        if len(labels) and s > 0.03:

            box = np.array([xmin, ymin, xmax, ymax], dtype=np.float32)

            ioa = bbox\_ioa(box, labels[:, 1:5])  # intersection over area

            labels = labels[ioa < 0.60]  # remove >60% obscured labels

    return labels

def create\_folder(path='./new'):

    # Create folder

    if os.path.exists(path):

        shutil.rmtree(path)  # delete output folder

    os.makedirs(path)  # make new output folder

def flatten\_recursive(path='../coco128'):

    # Flatten a recursive directory by bringing all files to top level

    new\_path = Path(path + '\_flat')

    create\_folder(new\_path)

    for file in tqdm(glob.glob(str(Path(path)) + '/\*\*/\*.\*', recursive=True)):

        shutil.copyfile(file, new\_path / Path(file).name)

def extract\_boxes(path='../coco128/'):  # from utils.datasets import \*; extract\_boxes('../coco128')

    # Convert detection dataset into classification dataset, with one directory per class

    path = Path(path)  # images dir

    shutil.rmtree(path / 'classifier') if (path / 'classifier').is\_dir() else None  # remove existing

    files = list(path.rglob('\*.\*'))

    n = len(files)  # number of files

    for im\_file in tqdm(files, total=n):

        if im\_file.suffix[1:] in img\_formats:

            # image

            im = cv2.imread(str(im\_file))[..., ::-1]  # BGR to RGB

            h, w = im.shape[:2]

            # labels

            lb\_file = Path(img2label\_paths([str(im\_file)])[0])

            if Path(lb\_file).exists():

                with open(lb\_file, 'r') as f:

                    lb = np.array([x.split() for x in f.read().strip().splitlines()], dtype=np.float32)  # labels

                for j, x in enumerate(lb):

                    c = int(x[0])  # class

                    f = (path / 'classifier') / f'{c}' / f'{path.stem}\_{im\_file.stem}\_{j}.jpg'  # new filename

                    if not f.parent.is\_dir():

                        f.parent.mkdir(parents=True)

                    b = x[1:] \* [w, h, w, h]  # box

                    # b[2:] = b[2:].max()  # rectangle to square

                    b[2:] = b[2:] \* 1.2 + 3  # pad

                    b = xywh2xyxy(b.reshape(-1, 4)).ravel().astype(np.int)

                    b[[0, 2]] = np.clip(b[[0, 2]], 0, w)  # clip boxes outside of image

                    b[[1, 3]] = np.clip(b[[1, 3]], 0, h)

                    assert cv2.imwrite(str(f), im[b[1]:b[3], b[0]:b[2]]), f'box failure in {f}'

def autosplit(path='../coco128', weights=(0.9, 0.1, 0.0), annotated\_only=False):

    """ Autosplit a dataset into train/val/test splits and save path/autosplit\_\*.txt files

    Usage: from utils.datasets import \*; autosplit('../coco128')

    Arguments

        path:           Path to images directory

        weights:        Train, val, test weights (list)

        annotated\_only: Only use images with an annotated txt file

    """

    path = Path(path)  # images dir

    files = sum([list(path.rglob(f"\*.{img\_ext}")) for img\_ext in img\_formats], [])  # image files only

    n = len(files)  # number of files

    indices = random.choices([0, 1, 2], weights=weights, k=n)  # assign each image to a split

    txt = ['autosplit\_train.txt', 'autosplit\_val.txt', 'autosplit\_test.txt']  # 3 txt files

    [(path / x).unlink() for x in txt if (path / x).exists()]  # remove existing

    print(f'Autosplitting images from {path}' + ', using \*.txt labeled images only' \* annotated\_only)

    for i, img in tqdm(zip(indices, files), total=n):

        if not annotated\_only or Path(img2label\_paths([str(img)])[0]).exists():  # check label

            with open(path / txt[i], 'a') as f:

                f.write(str(img) + '\n')  # add image to txt file

helpers.py blurring images

def mosaic\_blur(image, rect, step=20):

    h, w = image.shape[:2]

    for i in range(rect[0]+step, rect[2], step):

        for j in range(rect[1]+step, rect[3], step):

            image[i-step:i, j-step:j] = image[i,j]

    return image

def mosaic\_blur\_multiple(image, rects, step=20):

    for rect in rects:

        rect = [int(r) for r in rect]

        for i in range(rect[1]+step, rect[3], step):

            for j in range(rect[0]+step, rect[2], step):

                image[i-step:i, j-step:j] = image[i,j]

    return image