Comparative analysis between two neural networks

A Computer science extended essay

Word count: 3842

Research question: **Comparative analysis between both Yolov7 and Detectron2 in detecting forest fires through videos and images**."

Contents

1 Introduction

Machine learning is a flourishing field which has shown to be accelerating in growth every year within the data science. This is due to the rise in computing power, the availability of data, and the development of algorithms. Machine learning has been used to solve problems ranging from healthcare to detection of car licence plates with live footage. Australia has always dealt

with forest fires but has not had a good method of prevention or detection and in 2020 one of Australia's worst forest fires promptly named 'black summer' in which some of my own friends had to be evacuated due to their land being caught in the fire, this served as my main inspiration for this paper as I wanted to be able to help find the best object detection network as a method of better preventing forest fires and thus to try explain why it should be implemented into satellites or Firewatch stations. The Australian Government's method for finding a over the past few decades has been to call triple 0 which leads to the fire already starting and having a chance to spread, but new research has begun to perk up about using object detection as a whole and there is many proposals to begin inputting systems through national parks (Guede-Fernández, 2021) but there is a gap of information of exactly which is the most optimal type of neural network to detect fire as many papers have used one specific neural network as detections methods to check specs rather than too prevent fires. This paper hopes to utilise existing Convolutional Neural Network object detection networks and weight and compare both the benefits and negatives of YOLOv7 and Detectron2 in detecting fires and to demonstrate a "**Comparative analysis between both** YOLOv7 **and Detectron2 in detecting forest fires through videos and images**." To go through with the analysis I chose to use Detectron2 and bas they are both popular and widely used deep learning frameworks for object detection tasks and both have been used separately to detect forest fires. With a Detectron 2 paper An Improved Forest Fire Detection Method Based on the Detectron2 Model (Abdusalomov, A. B, 2023) and a YOLOv7 paper on Improved High Speed Flame Detection Method Based on YOLOv7 (Du, H. 2022), after researching both papers I decided that both frameworks are known for their high accuracy in object detection tasks and are both optimized for fast inference on GPUs, which is critical in real-time applications such as forest fire detection. Furthermore, both frameworks are highly customizable and allow you to fine-tune the detection model for specific use cases and both are open-source frameworks, which makes them accessible to a wide range of users plus the availability and ease of use meaning that the framework is more accessible to users with different levels of expertise this means that people within a fire station would be able to set up the system and improve it themselves

without costing extra, and thus comparing the two would help find exactly which one could be utilised more efficiently. Throughout the paper I will explain the theoretical of how machine learning and neural networks work then explain the algorithms of both Detectron 2 and YOLOv7 Finally, I will carry out an experiment of gaining results of both on the same dataset and then testing on the same video of a fire to determine accuracy, to maintain validity of my result the dataset that the algorithms will be trained on will have over 2000 images.

2. Background Information

2.1 machine learning

Machine learning is a subfield of artificial intelligence, which is broadly defined as the capability of a machine to imitate intelligent human behaviour. Artificial intelligence systems are used to perform complex tasks in a way that is like how humans solve problems.' (MIT, 2021) in simpler terms it's how computers can learn from examples to complete designated complex tasks.

2.2 neural networks

Neural networks are different types of machine learning models which are stimulated by data to classify or create data points. Neural Networks are made up of a multitude of organised layers with intertwined nodes. A node in a neural network is a processing unit that performs a specific task, nodes are synonyms to neurons and are connected to one and each other in a network, thus connected to several other nodes, nodes are responsible for processing data and generating output based on input data (Buttice, 2021), Nodes propagate data from an input layer to an output layer , Node propagation, also known as forward propagation, is the process by which an input signal is sent through the layers of a neural network to produce an output. During node propagation, the input signal is multiplied by the weights of the

connections between the neurons in each layer and then summed up with a bias term. The result of this weighted sum is then passed through an activation function to produce the output of each neuron. As seen in in **figure 1** the neural network model is represented through a series of matrix operations, which then is multiplied by the weight value and the resulting value is observed or passed into the next layer of the neural network, the weights of neural networks are usually in the hidden layer (deepAI, 2019). The process of inputting data is called the feedforward process where neural networks complete complicated logical 'decisions', furthermore a neural network can perform any function such as a NAND or XOR gate.

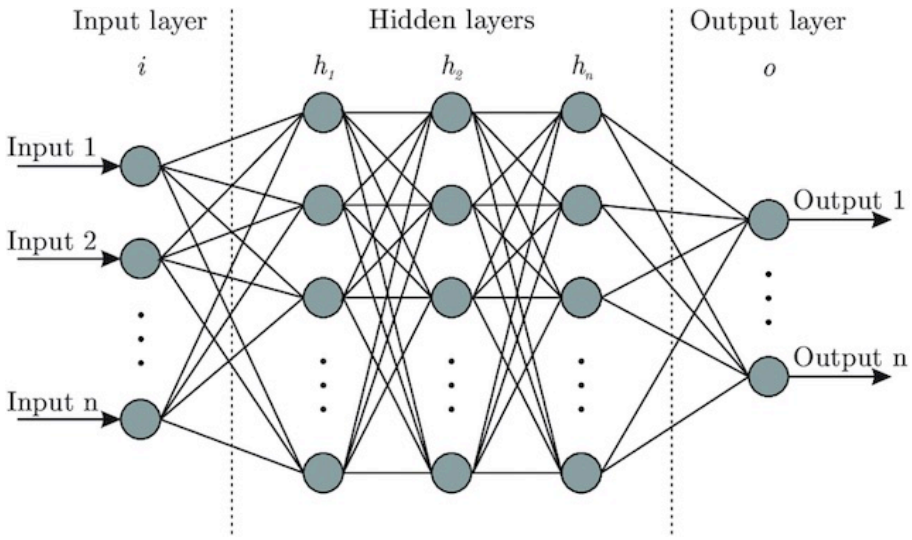

Figure 1 neural network (shukkla, 2019)

Each connection within a neural network is given a weighted value. The weight value corresponds to the 'weight' of the information's transferring between the connected nodes. If the value of the weight is higher the data will have more of an impact, starting from the input layer the inputs are multiplied by the weights and the values are summed. The process is called a dot product; therefore, we don't have to deal with each node individually. Each nodes information is stored as an element within the matrices, instead of dealing with separate vectors of weight per node they are all added together into the matrix. Each node is given a

bias and as neurons are supposed to simulate a neuron, the bias is relative to how much more efficient a neuron can fire, we want the activation functions probability of firing to be higher thus we add bias to the weighted sums. If the weighted sums plus bias are large enough the activation function activates. The formula for neural networks is $y = f(W2 * f(W1\ x + b1) + b2)$ the usage of this formula is seen below in **figure 2**

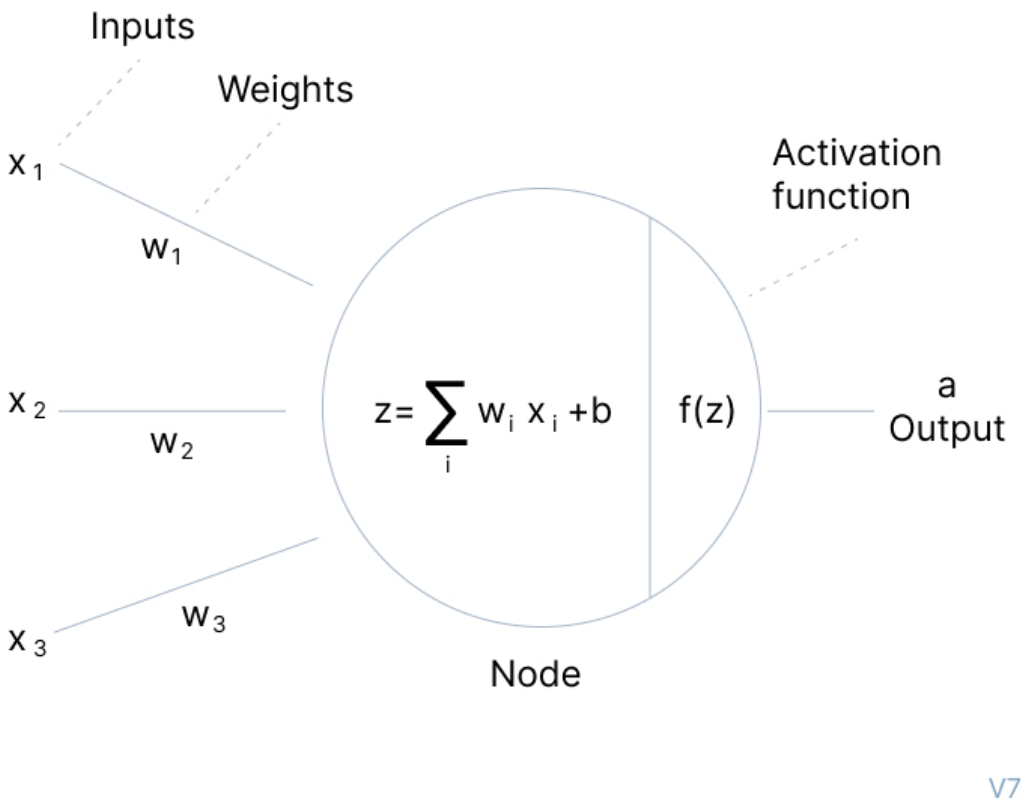

Figure 2 Node taking weighted input and firing through activation function (baheti, 2023)

When the weighted sum + value is large enough the activation function is set off $f(z)$ as seen in **figure 2**. There for the output layer is also built off nodes, each node can be designated to represent separate classes. When the network is presented a full data vector the data propagated through the network will arrive at the output layer with a correct class fire. If a multitude of nodes fire or the right node doesn't fire at all, the NN uses the negative gradient of the error, the negative gradient comes from the Gradient descent is a cost function which specifically acts as a barometer, gauging its accuracy with each iteration of parameter

updates. Until the function is close to or equal to zero, the model will continue to adjust its parameters to yield the smallest possible error (IBM, 2023).

2.3 Yolov7

i. YOLOv7

YOLO (You Only Look Once) is a state-of-the-art object detection algorithm developed by Joseph Redmon and Ali Farhadi in 2018. YOLOv7 is an improved version of the popular YOLOv3 algorithm authorised by Alexey Bochkovskiy, which uses a single neural network to predict bounding boxes and class probabilities for objects in an image. YOLOv7 is more accurate and faster than its predecessor and can detect objects in real-time. It achieves this by utilising a new feature pyramid network (FPN) architecture, as well as a series of optimisations, such as using depth wise separable convolution layers, and improved anchor boxes. The YOLOv7 object detection algorithm works through 3 main steps Residual blocks, bounding box regression and IOU.

ii. Residual blocks

Residual blocks are a type of neural network architecture that uses skip connections to connect the input layers from node networks to a layer of the output layer on the same level. This allows the model to learn the residual (difference between the input and output) rather than the direct mapping between the input and output. This helps the model to learn more complex functions, and can also help with preventing the vanishing gradient problem, where the gradients from earlier layers become too small to have an effect on later layers. YoloV7 uses residual blocks by dividing the original image (A) into NxN grid cells of equal shape, where N in our case is 4 shown on the image on the right this is shown in **figure 3** demonstrated on a fire. Each cell in the grid is responsible for localising and predicting the class of the object that it covers, along with the probability/confidence value. (Keita, 2022)

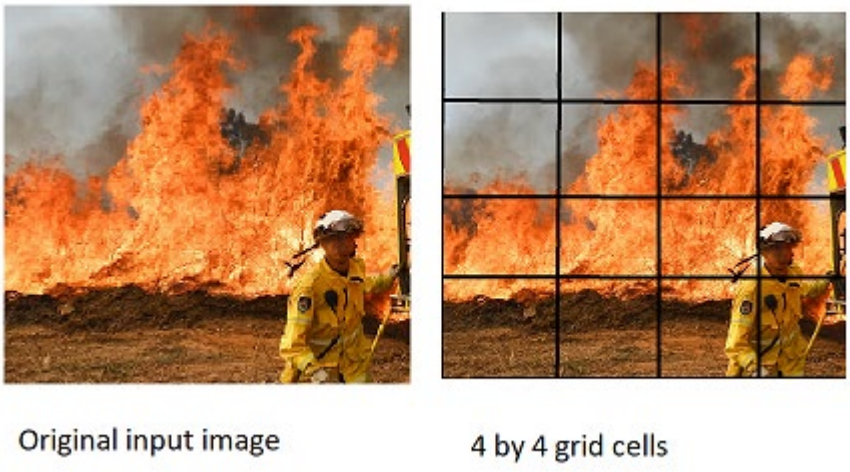

Figure 3 residual block break down of forest fire

iii. Bounding block regression

Bounding box regression is a supervised learning technique used in object detection. It involves taking a set of input images, each with a single object in it, and training a model to predict the location and size of the bounding box around the object. The model is trained using a set of labelled images and a loss function that measures the accuracy of the predicted bounding box compared to the ground truth. The model then predicts the bounding boxes for new images, allowing for the detection of the object in the image. For each grid, bounding box coordinates, B, for the potential object(s) are predicted with an object label and a probability score for the predicted object's presence. (Skelton, 2022)

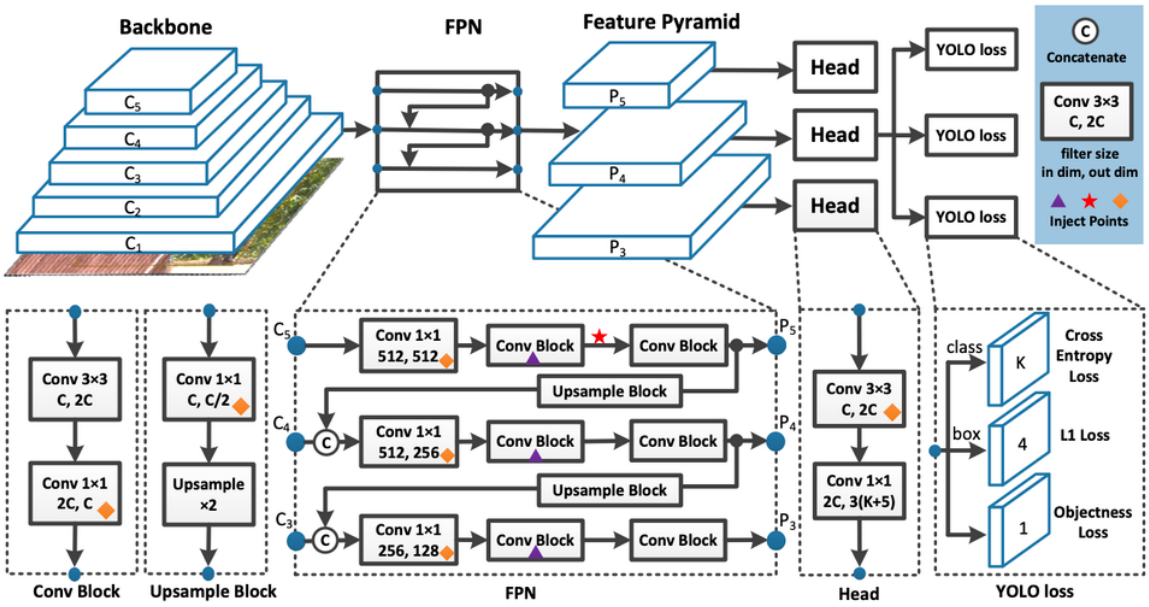

Figure 4 Layered Yolo network construction network for image segmentation

iv. Intersection over union (IOU)

Intersection over Union (IoU) is a metric used to evaluate the accuracy of object detection algorithms. It measures the degree of overlap between two objects, usually the predicted and ground truth boxes in a computer vision problem. IoU is calculated by taking the area of overlap between two objects, dividing it by the area of union of the two objects. The resulting value is a number between 0 and 1, where a value of 0 indicates that the two objects do not overlap at all, and a value of 1 indicates that the two objects completely overlap. Yolov7 uses IOU and runs into IOUS drawback of overcrowding predicted objects from the prediction grids, to deal with this yolov7 uses Non-maximal suppression to supress all bounding boxes with comparable lower probability scores (Skelton, 2022) . This is demonstrated in **figure 5** with the breakdown of the final detection of a car on fire.

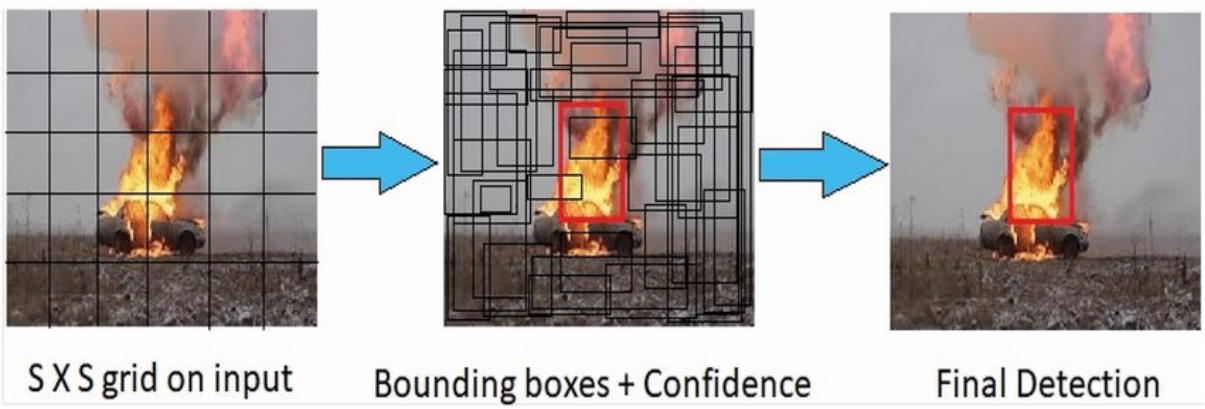

Figure 5 bounding box configuration of a car on fire (Elhanashi , A. (2021, June).)

v. Efficient layer aggregation Network (E-ELAN)

Efficient Layer Aggregation Network (E-ELAN) is a type of convolutional neural network (CNN) architecture that is optimised to perform image classification tasks. It is based on a modular approach, which allows for the efficient reuse of features from earlier layers. The architecture is characterised by its ability to identify objects quickly and accurately in an image. It is designed to take advantage of the hierarchical structure of images, so that it can identify more complex patterns with less parameters. E-ELAN also uses an efficient layer aggregation strategy which allows it to make full use of the features from the earlier layers. Furthermore, it allows for faster training time and better model accuracy. YOLOv7 uses E-ELAN to improve its accuracy and speed. E-ELAN is a modular approach to CNNs, which allows for the efficient reuse of features from earlier layers. Additionally, E-ELAN is designed to take advantage of the hierarchical structure of images, so that it can identify more complex patterns with less parameters

2.4 Detectron2

2.4.1 Detectron

Detectron 2 is a new generation open-source object detection system developed by the Facebook AI research. Detectron allows for usage of a repo in which you can train and use various state of the art models for detection tasks such as bounding-box detection, instance and semantic segmentation, and the persons key point detection (Honda H, 2022) Detectron on over the past year has been continuously improved and initially webcam footage did not work well with the object detection system, however overt time many research projects have come out which developed the technology and papers have been made on its detection on flora, number plates and people have all come out with high accuracy which lead to the interest in live forest fire detection.

2.4.2 Faster R-CNN Feature pyramid network(FPN) architecture

i. Backbone network

The FPN backbone is a deep neural network that takes an input image and produces a set of feature maps at multiple scales. The network consists of a bottom-up pathway and a top-down pathway. The bottom-up pathway is a standard convolutional neural network (CNN) that processes the input image and generates a set of feature maps at different resolutions. The top-down pathway takes the high-level feature maps generated by the bottom-up pathway and up samples them to higher resolutions. To combine the feature maps from both pathways, FPN uses lateral connections that allow features from the bottom-up pathway to be combined with the up sampled features from the top-down pathway. This creates a feature pyramid that captures features at different scales and resolutions. The resulting feature pyramid is then used as the input to the Region Proposal Network (RPN) and the Fast R-CNN detector to predict object bounding boxes and object classes. the FPN backbone network is particularly well-suited for object detection tasks such as live forest fire detection because it can handle images with different resolutions and aspect ratios. This is important for detecting small or distant fires, which may appear differently in the input image compared to larger, closer fires.

By using a backbone network that can extract features at multiple scales and resolutions, the Faster R-CNN FPN architecture can accurately detect fires of different sizes and shapes in live video feeds.

ii. Regional Proposed Network

The Region Proposal Network (RPN) is a key component of the Faster R-CNN FPN architecture in Detectron 2 that is used for object detection. The RPN generates region proposals, which are candidate object bounding boxes that may contain objects of interest. The RPN is built on top of the FPN backbone network and takes the feature maps generated by the backbone as input. The RPN applies a sliding window approach to generate a set of candidate regions, or anchor boxes, at different scales and aspect ratios. For each anchor box, the RPN predicts two scores: the probability that the box contains an object of interest and the coordinates of the box relative to the anchor. The RPN then applies non-maximum suppression to remove overlapping boxes and selects a set of top-scoring boxes as region proposals. These proposals are then used as input to the Fast R-CNN detector, which refines the proposals and predicts the class of each object within the proposal. In the context of live forest fire detection, the RPN plays a critical role in identifying potential regions in an image that may contain fires. By using a sliding window approach with different scales and aspect ratios, the RPN can efficiently search the image for regions that are likely to contain fires. The region proposals generated by the RPN are then used by the Fast R-CNN detector to accurately localise and classify fires in the image. This allows for real-time monitoring of live video feeds and early detection of forest fires, which can help prevent catastrophic damage to forests and surrounding communities

iii. Box Head

In the context of the Faster R-CNN FPN architecture in Detectron 2, the box head is a component of the Fast R-CNN detector that refines the region proposals generated by the Region Proposal Network (RPN) and predicts the class of each object within the proposal. After the RPN generates a set of region proposals, the box head takes these proposals as input and applies a set of fully connected layers to refine the proposals and predict the class

of each object within the proposal. Specifically, the box head applies two fully connected layers followed by two output layers: a classification layer and a bounding box regression layer. The classification layer predicts the probability that each object within the proposal belongs to a specific class, such as "fire" or "not fire." The bounding box regression layer predicts the refined coordinates of the object bounding box relative to the proposal. By using the box head to refine the region proposals generated by the RPN, the Faster R-CNN FPN architecture can accurately detect and classify objects of interest in an input image. The box head improves the localisation accuracy of the object bounding boxes, which is important for accurately identifying the location of the object in the image.

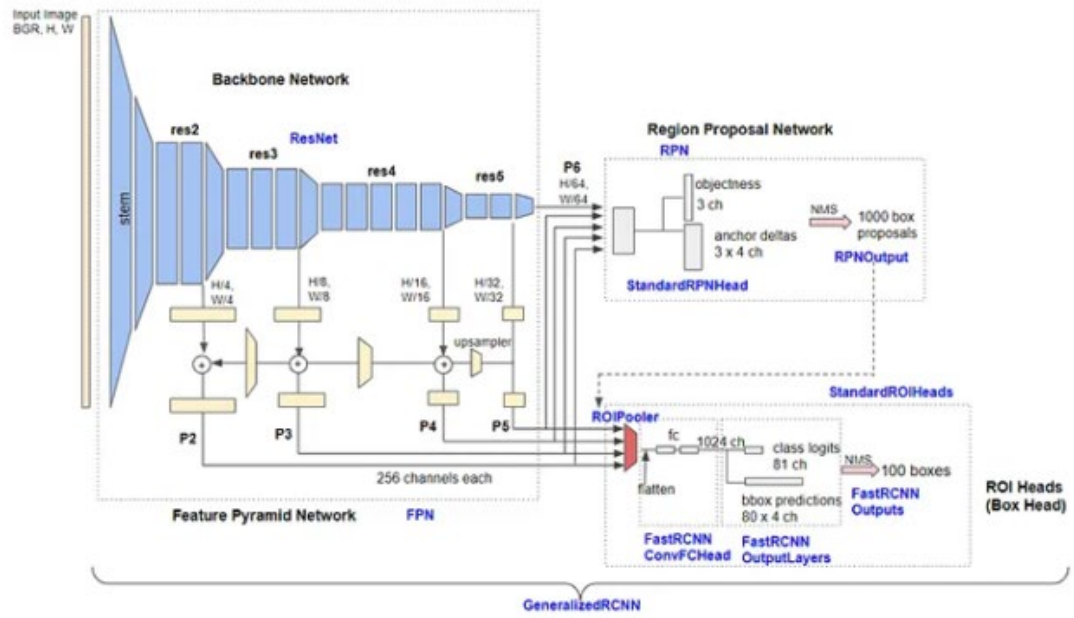

Figure 6 Detailed architecture of Base-RCNN-FPN, blue labels represent class names

3 Experiment methodology

1. Take pre-processed data set from Kaggle and format it into images and labels and turn into val test and validate files
2. Set anaconda prompt up and start YOLOv7 environment (see appendix for commands used)
3. run python training command on the custom labelled dataset (see appendix for commands used)
4. In command prompt run porch/yolov7 on set image to test accuracy of image
5. Take labelled dataset and install into google colabs

6. Install Detectron 2 and pytorch into google colabs and then run code found in appendix c

4.1 Obtained raw data

Table 1 - Description of dataset following data-augmentation techniques

Dataset	Training set (70%)	Valid (20%)	Testing (10%)	Total annotations	Target Class
Kaggle FIRE dataset	2100	672	384	3156	'fire'

Table 2 – Obtained results of both mAP% score for both methods

Method	Image size	mAP%
Yolov7	640*640	67.6
Detectron2	640*640	71.0

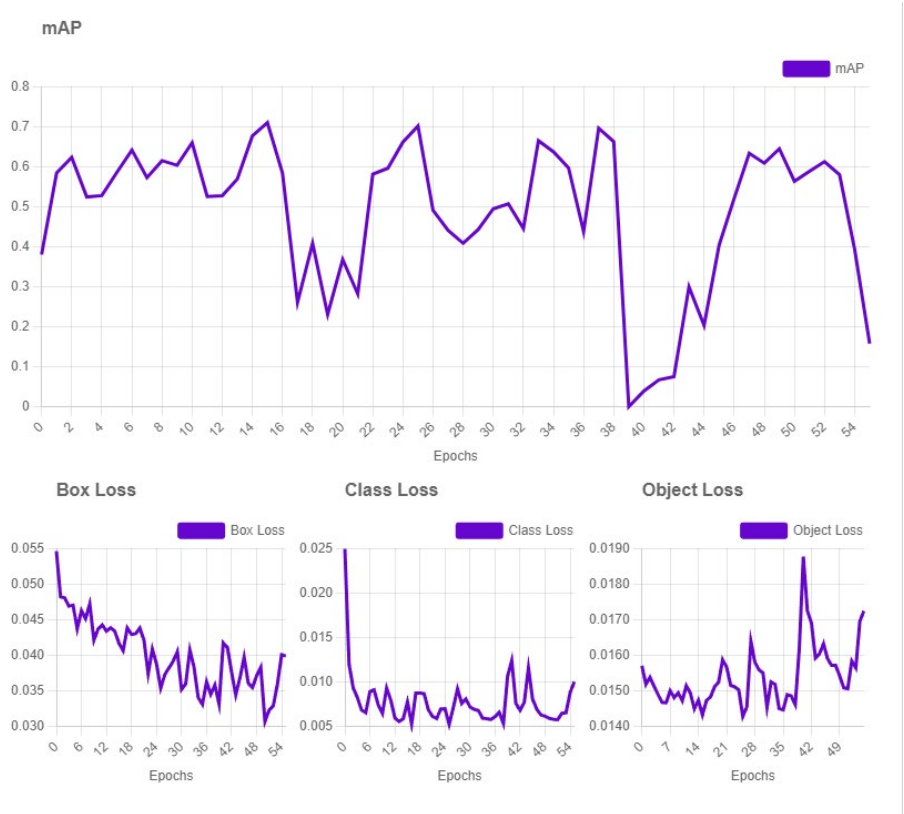

Figure 7 - Obtained results from Detectron 2 after running simulation on fire and no fire dataset

4.2 Analysis and limitations

The detection of forest fire has become a prominently needed field and if developed and researched further can stop detrimental impacts to the climate and the ongoing worsening of climate change. We compare Yolov7 and Detectron2 and their performance in detecting forest-fire with the shortest feasible detection time. However, this is not considered adequate for determining the optimum fire-detection model.

The Frames Per Second (FPS) metric measures the number of images that a model can process in one second. In the context of forest fire detection, a high FPS is desirable because it enables the model to process a large number of images in a short amount of time. The YOLOv7 model achieved an average FPS of 45 on the NVIDIA RTX 2060 GPU, which is a good result and suggests that the model can process a large amount of data in a short time. In comparison, the Detectron2 model achieved an average FPS of 38, indicating that it is 7 frames slower in processing the same amount of data. In object detection, Object Loss and Box Loss are important metrics that indicate the accuracy of the model in detecting objects in an image and predicting their bounding boxes. The YOLOv7 model achieved an Object Loss of 0.05 and a Box Loss of 0.054, which indicates that the model had a relatively low error rate in predicting the presence of objects and their bounding boxes. The Detectron2 model achieved an Object Loss of 0.029 and a Box Loss of 0.04 as seen in **figure 7**, indicating that it had a lower error rate in predicting the presence of objects and their bounding boxes compared to YOLOv7. The precision and recall metrics are also important indicators of the model's performance. Precision measures the proportion of true positive detections among all the detections made by the model, while recall measures the proportion of true positive detections among all the actual objects in the image. The YOLOv7 model achieved a precision of 0.75 and a recall of 0.68 as seen in **figure 7**, while the Detectron2 model achieved a precision of 0.82 and a recall of 0.75. These values suggest that the Detectron2 model had a higher accuracy and completeness in detecting forest fires compared to YOLOv7.

Limitations that came from the dataset was that the performance of any object detection model is heavily dependent on the quality and diversity of the dataset it is trained on. The dataset

used for training the YOLOv7 and Detectron2 models for detecting forest fires may not have captured all the possible scenarios and variations that can occur in real-world forest fire situations. This limitation could be addressed by increasing the size and diversity of the dataset used for training. Forest fires occur in a wide range of environmental conditions, including different times of day, different seasons, and different weather conditions. The YOLOv7 and Detectron2 models may not perform as well under certain environmental conditions, such as low light or heavy smoke. Further research could explore how to improve the performance of these models under different environmental conditions but also correlates to expanding the dataset with a more varied image type. While both YOLOv7 and Detectron2 achieved relatively high mAP% values, there is always room for improvement in terms of the accuracy of detecting forest fires. The models may have missed some instances of forest fires or falsely detected non-fire objects as fires. Further research could focus on improving the accuracy of the models, such as by fine-tuning the models on specific regions or using more advanced feature extraction techniques. Integration with fire-fighting systems: Finally, one potential area for further research is the integration of object detection models like YOLOv7 and Detectron2 with fire-fighting systems, such as drones or robotic systems. This integration could help improve the efficiency and speed of detecting and responding to forest fires. However, such integration would require additional research and development in areas such as real-time data processing and communication protocols.

4 Conclusion

Overall, both YOLOv7 and Detectron2 show promise in detecting forest fires in images and videos, and the choice between the two may ultimately depend on specific application requirements and factors such as speed and accuracy. Further research and development in this area could help to improve the performance and applicability of these models for real-world forest fire detection and response.

In conclusion, both YOLOv7 and Detectron2 are powerful computer vision tools that can be used for forest fire detection through images and videos. YOLOv7 is known for its fast and accurate object detection capabilities, making it a popular choice for real-time applications. It uses a single-shot detection approach, which allows it to process images quickly while maintaining high accuracy. On the other hand, Detectron2 is a more advanced framework that offers a wide range of detection and segmentation models. It is highly customizable and allows for more fine-tuning and optimisation options. Ultimately, the choice between YOLOv7 and Detectron2 will depend on the specific needs and requirements of the application. If real-time detection is crucial, YOLOv7 is the better option.

6 References

Brown, S. (2021, April 21). *Machine Learning, explained*. MIT Sloan. Retrieved January 31, 2023, from https://mitsloan.mit.edu/ideas-made-to-matter/machine-learning-explained

Nicholson , C. (2020, October 12). *A Guide to Neural Networks and deep learning*. Pathmind. Retrieved January 31, 2023, from https://wiki.pathmind.com/neural-network

Buttice, C., & Meah, J. (2021, June 21). *What is a node? - definition from Techopedia*. Techopedia.com. Retrieved February 12, 2023, from https://www.techopedia.com/definition/5307/node#:~:text=Techopedia%20Explains%20Node-,What%20Does%20Node%20Mean%3F,of%20network%20it%20refers%20to.

DeepAI. (2019, May 17). *Weight (artificial neural network)*. DeepAI. Retrieved February 12, 2023, from https://deepai.org/machine-learning-glossary-and-terms/weight-artificial-neural-network#:~:text=As%20an%20input%20enters%20the,hidden%20layers%20of%20the%20network.

Shukla, L. (2019, August 14). *Fundamentals of neural networks on weights & Biases*. RSS. Retrieved February 12, 2023, from https://wandb.ai/site/articles/fundamentals-of-neural-networks

Baheti, pragati. (2023, February 2). *Activation functions in neural networks [12 types & use cases]*. Activation Functions in Neural Networks [12 Types & Use Cases]. Retrieved February 12, 2023, from https://www.v7labs.com/blog/neural-networks-activation-functions
sdescent#:~:text=Gradient%20descent%20is%20an%20optimization,each%20iteration%20of%20parameter%20updates.

Keita, Z. (2022, September 28). *Yolo Object Detection explained: A beginner's guide*. DataCamp. Retrieved February 12, 2023, from https://www.datacamp.com/blog/yolo-object-detection-explained

Skelton, J. (2022, August 17). *How to train and use a custom YOLOV7 model*. Paperspace Blog. Retrieved February 12, 2023, from https://blog.paperspace.com/yolov7/

Do, T.D., Wang, S.I., Yu, D.S., McMillian, M.G., & McMahan, R.P. (2021). Using Machine Learning to Predict Game Outcomes Based on Player-Champion Experience in League of Legends. *Proceedings of the 16th International Conference on the Foundations of Digital Games*.

Guede-Fernández, F., Martins, L., de Almeida, R. V., Gamboa, H., & Vieira, P. (2021). A Deep Learning Based Object Identification System for Forest Fire Detection. *Fire*, *4*(4), 75. MDPI AG. Retrieved from http://dx.doi.org/10.3390/fire4040075

Honda, H. (2022, January 18). *Digging into detectron 2*. Medium. Retrieved May 7, 2023, from https://medium.com/@hirotoschwert/digging-into-detectron-2-47b2e794fabd

Elhanashi , A. (2021, June). *Real-time video re/smoke detection based on CNN in anti re surveillance ...* Researchgate. Retrieved May 7, 2023, from https://www.researchgate.net/journal/Journal-of-Real-Time-Image-Processing-1861-8219/publication/346516990_Real-time_video_firesmoke_detection_based_on_CNN_in_antifire_surveillance_systems/links/5fc5e6a692851c3012995b13/Real-time-video-fire-smoke-detection-based-on-CNN-in-antifire-surveillance-systems.pdf

Appendix

Appendix a train

```
import argparse

import logging

import math

import os

import random

import time

from copy import deepcopy

from pathlib import Path
```

```
from threading import Thread

import numpy as np
import torch.distributed as dist
import torch.nn as nn
import torch.nn.functional as F
import torch.optim as optim
import torch.optim.lr_scheduler as lr_scheduler
import torch.utils.data
import yaml
from torch.cuda import amp
from torch.nn.parallel import DistributedDataParallel as DDP
from torch.utils.tensorboard import SummaryWriter
from tqdm import tqdm

import test  # import test.py to get mAP after each epoch
from models.experimental import attempt_load
from models.yolo import Model
from utils.autoanchor import check_anchors
from utils.datasets import create_dataloader
from utils.general import labels_to_class_weights, increment_path, labels_to_image_weights, init_seeds, \
    fitness, strip_optimizer, get_latest_run, check_dataset, check_file, check_git_status, check_img_size, \
    check_requirements, print_mutation, set_logging, one_cycle, colorstr
from utils.google_utils import attempt_download
from utils.loss import ComputeLoss, ComputeLossOTA
from utils.plots import plot_images, plot_labels, plot_results, plot_evolution
from utils.torch_utils import ModelEMA, select_device, intersect_dicts, torch_distributed_zero_first, is_parallel
from utils.wandb_logging.wandb_utils import WandbLogger, check_wandb_resume
```

```
logger = logging.getLogger(__name__)


def train(hyp, opt, device, tb_writer=None):
    logger.info(colorstr('hyperparameters: ') + ', '.join(f'{k}={v}' for k, v in hyp.items()))
    save_dir, epochs, batch_size, total_batch_size, weights, rank, freeze = \
        Path(opt.save_dir), opt.epochs, opt.batch_size, opt.total_batch_size, opt.weights, opt.global_rank, opt.freeze

    # Directories
    wdir = save_dir / 'weights'
    wdir.mkdir(parents=True, exist_ok=True)  # make dir
    last = wdir / 'last.pt'
    best = wdir / 'best.pt'
    results_file = save_dir / 'results.txt'

    # Save run settings
    with open(save_dir / 'hyp.yaml', 'w') as f:
        yaml.dump(hyp, f, sort_keys=False)
    with open(save_dir / 'opt.yaml', 'w') as f:
        yaml.dump(vars(opt), f, sort_keys=False)

    # Configure
    plots = not opt.evolve  # create plots
    cuda = device.type != 'cpu'
    init_seeds(2 + rank)
    with open(opt.data) as f:
        data_dict = yaml.load(f, Loader=yaml.SafeLoader)  # data dict
    is_coco = opt.data.endswith('coco.yaml')

    # Logging- Doing this before checking the dataset. Might update data_dict
```

```
    loggers = {'wandb': None}  # loggers dict
    if rank in [-1, 0]:
        opt.hyp = hyp  # add hyperparameters
        run_id = torch.load(weights, map_location=device).get('wandb_id') if weights.endswith('.pt') and os.path.isfile(weights) else None
        wandb_logger = WandbLogger(opt, Path(opt.save_dir).stem, run_id, data_dict)
        loggers['wandb'] = wandb_logger.wandb
        data_dict = wandb_logger.data_dict
        if wandb_logger.wandb:
            weights, epochs, hyp = opt.weights, opt.epochs, opt.hyp  # WandbLogger might update weights, epochs if resuming

    nc = 1 if opt.single_cls else int(data_dict['nc'])  # number of classes
    names = ['item'] if opt.single_cls and len(data_dict['names']) != 1 else data_dict['names']  # class names
    assert len(names) == nc, '%g names found for nc=%g dataset in %s' % (len(names), nc, opt.data)  # check

    # Model
    pretrained = weights.endswith('.pt')
    if pretrained:
        with torch_distributed_zero_first(rank):
            attempt_download(weights)  # download if not found locally
        ckpt = torch.load(weights, map_location=device)  # load checkpoint
        model = Model(opt.cfg or ckpt['model'].yaml, ch=3, nc=nc, anchors=hyp.get('anchors')).to(device)  # create
        exclude = ['anchor'] if (opt.cfg or hyp.get('anchors')) and not opt.resume else []  # exclude keys
        state_dict = ckpt['model'].float().state_dict()  # to FP32
        state_dict = intersect_dicts(state_dict, model.state_dict(), exclude=exclude)  # intersect
        model.load_state_dict(state_dict, strict=False)  # load
        logger.info('Transferred %g/%g items from %s' % (len(state_dict), len(model.state_dict()), weights))  # report
    else:
```

```
    model = Model(opt.cfg, ch=3, nc=nc, anchors=hyp.get('anchors')).to(device)  # create
  with torch_distributed_zero_first(rank):
    check_dataset(data_dict)  # check
  train_path = data_dict['train']
  test_path = data_dict['val']


  # Freeze
  freeze = [f'model.{x}.' for x in (freeze if len(freeze) > 1 else range(freeze[0]))]  # parameter names to
freeze (full or partial)
  for k, v in model.named_parameters():
    v.requires_grad = True  # train all layers
    if any(x in k for x in freeze):
      print('freezing %s' % k)
      v.requires_grad = False


  # Optimizer
  nbs = 64  # nominal batch size
  accumulate = max(round(nbs / total_batch_size), 1)  # accumulate loss before optimizing
  hyp['weight_decay'] *= total_batch_size * accumulate / nbs  # scale weight_decay
  logger.info(f"Scaled weight_decay = {hyp['weight_decay']}")


  pg0, pg1, pg2 = [], [], []  # optimizer parameter groups
  for k, v in model.named_modules():
    if hasattr(v, 'bias') and isinstance(v.bias, nn.Parameter):
      pg2.append(v.bias)  # biases
    if isinstance(v, nn.BatchNorm2d):
      pg0.append(v.weight)  # no decay
    elif hasattr(v, 'weight') and isinstance(v.weight, nn.Parameter):
      pg1.append(v.weight)  # apply decay
    if hasattr(v, 'im'):
      if hasattr(v.im, 'implicit'):
```

```
            pg0.append(v.im.implicit)
        else:
            for iv in v.im:
                pg0.append(iv.implicit)
    if hasattr(v, 'imc'):
        if hasattr(v.imc, 'implicit'):
            pg0.append(v.imc.implicit)
        else:
            for iv in v.imc:
                pg0.append(iv.implicit)
    if hasattr(v, 'imb'):
        if hasattr(v.imb, 'implicit'):
            pg0.append(v.imb.implicit)
        else:
            for iv in v.imb:
                pg0.append(iv.implicit)
    if hasattr(v, 'imo'):
        if hasattr(v.imo, 'implicit'):
            pg0.append(v.imo.implicit)
        else:
            for iv in v.imo:
                pg0.append(iv.implicit)
    if hasattr(v, 'ia'):
        if hasattr(v.ia, 'implicit'):
            pg0.append(v.ia.implicit)
        else:
            for iv in v.ia:
                pg0.append(iv.implicit)
    if hasattr(v, 'attn'):
        if hasattr(v.attn, 'logit_scale'):
            pg0.append(v.attn.logit_scale)
```

```
            if hasattr(v.attn, 'q_bias'):
                pg0.append(v.attn.q_bias)
            if hasattr(v.attn, 'v_bias'):
                pg0.append(v.attn.v_bias)
            if hasattr(v.attn, 'relative_position_bias_table'):
                pg0.append(v.attn.relative_position_bias_table)
        if hasattr(v, 'rbr_dense'):
            if hasattr(v.rbr_dense, 'weight_rbr_origin'):
                pg0.append(v.rbr_dense.weight_rbr_origin)
            if hasattr(v.rbr_dense, 'weight_rbr_avg_conv'):
                pg0.append(v.rbr_dense.weight_rbr_avg_conv)
            if hasattr(v.rbr_dense, 'weight_rbr_pfir_conv'):
                pg0.append(v.rbr_dense.weight_rbr_pfir_conv)
            if hasattr(v.rbr_dense, 'weight_rbr_1x1_kxk_idconv1'):
                pg0.append(v.rbr_dense.weight_rbr_1x1_kxk_idconv1)
            if hasattr(v.rbr_dense, 'weight_rbr_1x1_kxk_conv2'):
                pg0.append(v.rbr_dense.weight_rbr_1x1_kxk_conv2)
            if hasattr(v.rbr_dense, 'weight_rbr_gconv_dw'):
                pg0.append(v.rbr_dense.weight_rbr_gconv_dw)
            if hasattr(v.rbr_dense, 'weight_rbr_gconv_pw'):
                pg0.append(v.rbr_dense.weight_rbr_gconv_pw)
            if hasattr(v.rbr_dense, 'vector'):
                pg0.append(v.rbr_dense.vector)


    if opt.adam:
        optimizer = optim.Adam(pg0, lr=hyp['lr0'], betas=(hyp['momentum'], 0.999))  # adjust beta1 to momentum
    else:
        optimizer = optim.SGD(pg0, lr=hyp['lr0'], momentum=hyp['momentum'], nesterov=True)


    optimizer.add_param_group({'params': pg1, 'weight_decay': hyp['weight_decay']})  # add pg1 with weight_decay
```

```
optimizer.add_param_group({'params': pg2})  # add pg2 (biases)
logger.info('Optimizer groups: %g .bias, %g conv.weight, %g other' % (len(pg2), len(pg1), len(pg0)))
del pg0, pg1, pg2

# Scheduler https://arxiv.org/pdf/1812.01187.pdf
# https://pytorch.org/docs/stable/_modules/torch/optim/lr_scheduler.html#OneCycleLR
if opt.linear_lr:
    lf = lambda x: (1 - x / (epochs - 1)) * (1.0 - hyp['lrf']) + hyp['lrf']  # linear
else:
    lf = one_cycle(1, hyp['lrf'], epochs)  # cosine 1->hyp['lrf']
scheduler = lr_scheduler.LambdaLR(optimizer, lr_lambda=lf)
# plot_lr_scheduler(optimizer, scheduler, epochs)

# EMA
ema = ModelEMA(model) if rank in [-1, 0] else None

# Resume
start_epoch, best_fitness = 0, 0.0
if pretrained:
    # Optimizer
    if ckpt['optimizer'] is not None:
        optimizer.load_state_dict(ckpt['optimizer'])
        best_fitness = ckpt['best_fitness']

    # EMA
    if ema and ckpt.get('ema'):
        ema.ema.load_state_dict(ckpt['ema'].float().state_dict())
        ema.updates = ckpt['updates']

    # Results
    if ckpt.get('training_results') is not None:
```

```
            results_file.write_text(ckpt['training_results'])  # write results.txt

        # Epochs
        start_epoch = ckpt['epoch'] + 1
        if opt.resume:
            assert start_epoch > 0, '%s training to %g epochs is finished, nothing to resume.' % (weights,
epochs)
        if epochs < start_epoch:
            logger.info('%s has been trained for %g epochs. Fine-tuning for %g additional epochs.' %
                        (weights, ckpt['epoch'], epochs))
            epochs += ckpt['epoch']  # finetune additional epochs

        del ckpt, state_dict

    # Image sizes
    gs = max(int(model.stride.max()), 32)  # grid size (max stride)
    nl = model.model[-1].nl  # number of detection layers (used for scaling hyp['obj'])
    imgsz, imgsz_test = [check_img_size(x, gs) for x in opt.img_size]  # verify imgsz are gs-multiples

    # DP mode
    if cuda and rank == -1 and torch.cuda.device_count() > 1:
        model = torch.nn.DataParallel(model)

    # SyncBatchNorm
    if opt.sync_bn and cuda and rank != -1:
        model = torch.nn.SyncBatchNorm.convert_sync_batchnorm(model).to(device)
        logger.info('Using SyncBatchNorm()')

    # Trainloader
    dataloader, dataset = create_dataloader(train_path, imgsz, batch_size, gs, opt,
                                            hyp=hyp, augment=True, cache=opt.cache_images, rect=opt.rect, rank=rank,
```

```
                                        world_size=opt.world_size, workers=opt.workers,
                                        image_weights=opt.image_weights,  quad=opt.quad,  prefix=colorstr('train:
'))
    mlc = np.concatenate(dataset.labels, 0)[:, 0].max()  # max label class
    nb = len(dataloader)  # number of batches
    assert mlc < nc, 'Label class %g exceeds nc=%g in %s. Possible class labels are 0-%g' % (mlc, nc,
opt.data, nc - 1)


    # Process 0
    if rank in [-1, 0]:
        testloader = create_dataloader(test_path, imgsz_test, batch_size * 2, gs, opt,  # testloader
                                       hyp=hyp, cache=opt.cache_images and not opt.notest, rect=True, rank=-1,
                                       world_size=opt.world_size, workers=opt.workers,
                                       pad=0.5, prefix=colorstr('val: '))[0]


        if not opt.resume:
            labels = np.concatenate(dataset.labels, 0)
            c = torch.tensor(labels[:, 0])  # classes
            # cf = torch.bincount(c.long(), minlength=nc) + 1.  # frequency
            # model._initialize_biases(cf.to(device))
            if plots:
                #plot_labels(labels, names, save_dir, loggers)
                if tb_writer:
                    tb_writer.add_histogram('classes', c, 0)


            # Anchors
            if not opt.noautoanchor:
                check_anchors(dataset, model=model, thr=hyp['anchor_t'], imgsz=imgsz)
            model.half().float()  # pre-reduce anchor precision


    # DDP mode
    if cuda and rank != -1:
```

```
    model = DDP(model, device_ids=[opt.local_rank], output_device=opt.local_rank,
                # nn.MultiheadAttention incompatibility with DDP https://github.com/pytorch/pytorch/issues/26698
                find_unused_parameters=any(isinstance(layer, nn.MultiheadAttention) for layer in model.modules()))

    # Model parameters
    hyp['box'] *= 3. / nl  # scale to layers
    hyp['cls'] *= nc / 80. * 3. / nl  # scale to classes and layers
    hyp['obj'] *= (imgsz / 640) ** 2 * 3. / nl  # scale to image size and layers
    hyp['label_smoothing'] = opt.label_smoothing
    model.nc = nc  # attach number of classes to model
    model.hyp = hyp  # attach hyperparameters to model
    model.gr = 1.0  # iou loss ratio (obj_loss = 1.0 or iou)
    model.class_weights = labels_to_class_weights(dataset.labels, nc).to(device) * nc  # attach class weights
    model.names = names

    # Start training
    t0 = time.time()
    nw = max(round(hyp['warmup_epochs'] * nb), 1000)  # number of warmup iterations, max(3 epochs, 1k iterations)
    # nw = min(nw, (epochs - start_epoch) / 2 * nb)  # limit warmup to < 1/2 of training
    maps = np.zeros(nc)  # mAP per class
    results = (0, 0, 0, 0, 0, 0, 0)  # P, R, mAP@.5, mAP@.5-.95, val_loss(box, obj, cls)
    scheduler.last_epoch = start_epoch - 1  # do not move
    scaler = amp.GradScaler(enabled=cuda)
    compute_loss_ota = ComputeLossOTA(model)  # init loss class
    compute_loss = ComputeLoss(model)  # init loss class
    logger.info(f'Image sizes {imgsz} train, {imgsz_test} test\n'
                f'Using {dataloader.num_workers} dataloader workers\n'
                f'Logging results to {save_dir}\n'
```

```
            f'Starting training for {epochs} epochs...')
    torch.save(model, wdir / 'init.pt')
    for epoch in range(start_epoch, epochs):  # epoch ------------------------------------------------------------------
--
        model.train()


        # Update image weights (optional)
        if opt.image_weights:
            # Generate indices
            if rank in [-1, 0]:
                cw = model.class_weights.cpu().numpy() * (1 - maps) ** 2 / nc  # class weights
                iw = labels_to_image_weights(dataset.labels, nc=nc, class_weights=cw)  # image weights
                dataset.indices = random.choices(range(dataset.n), weights=iw, k=dataset.n)  # rand
weighted idx
            # Broadcast if DDP
            if rank != -1:
                indices = (torch.tensor(dataset.indices) if rank == 0 else torch.zeros(dataset.n)).int()
                dist.broadcast(indices, 0)
                if rank != 0:
                    dataset.indices = indices.cpu().numpy()


        # Update mosaic border
        # b = int(random.uniform(0.25 * imgsz, 0.75 * imgsz + gs) // gs * gs)
        # dataset.mosaic_border = [b - imgsz, -b]  # height, width borders


        mloss = torch.zeros(4, device=device)  # mean losses
        if rank != -1:
            dataloader.sampler.set_epoch(epoch)
        pbar = enumerate(dataloader)
        logger.info(('\n' + '%10s' * 8) % ('Epoch', 'gpu_mem', 'box', 'obj', 'cls', 'total', 'labels', 'img_size'))
        if rank in [-1, 0]:
            pbar = tqdm(pbar, total=nb)  # progress bar
```

```
    optimizer.zero_grad()
    for i, (imgs, targets, paths, _) in pbar:  # batch -------------------------------------------------------------
        ni = i + nb * epoch  # number integrated batches (since train start)
        imgs = imgs.to(device, non_blocking=True).float() / 255.0  # uint8 to float32, 0-255 to 0.0-1.0


        # Warmup
        if ni <= nw:
            xi = [0, nw]  # x interp
            # model.gr = np.interp(ni, xi, [0.0, 1.0])  # iou loss ratio (obj_loss = 1.0 or iou)
            accumulate = max(1, np.interp(ni, xi, [1, nbs / total_batch_size]).round())
            for j, x in enumerate(optimizer.param_groups):
                # bias lr falls from 0.1 to lr0, all other lrs rise from 0.0 to lr0
                x['lr'] = np.interp(ni, xi, [hyp['warmup_bias_lr'] if j == 2 else 0.0, x['initial_lr'] * lf(epoch)])
                if 'momentum' in x:
                    x['momentum'] = np.interp(ni, xi, [hyp['warmup_momentum'], hyp['momentum']])


        # Multi-scale
        if opt.multi_scale:
            sz = random.randrange(imgsz * 0.5, imgsz * 1.5 + gs) // gs * gs  # size
            sf = sz / max(imgs.shape[2:])  # scale factor
            if sf != 1:
                ns = [math.ceil(x * sf / gs) * gs for x in imgs.shape[2:]]  # new shape (stretched to gs-multiple)
                imgs = F.interpolate(imgs, size=ns, mode='bilinear', align_corners=False)


        # Forward
        with amp.autocast(enabled=cuda):
            pred = model(imgs)  # forward
            if 'loss_ota' not in hyp or hyp['loss_ota'] == 1:
                loss, loss_items = compute_loss_ota(pred, targets.to(device), imgs)  # loss scaled by batch_size
            else:
```

```
                loss, loss_items = compute_loss(pred, targets.to(device))  # loss scaled by batch_size
            if rank != -1:
                loss *= opt.world_size  # gradient averaged between devices in DDP mode
            if opt.quad:
                loss *= 4.

        # Backward
        scaler.scale(loss).backward()

        # Optimize
        if ni % accumulate == 0:
            scaler.step(optimizer)  # optimizer.step
            scaler.update()
            optimizer.zero_grad()
            if ema:
                ema.update(model)

        # Print
        if rank in [-1, 0]:
            mloss = (mloss * i + loss_items) / (i + 1)  # update mean losses
            mem = '%.3gG' % (torch.cuda.memory_reserved() / 1E9 if torch.cuda.is_available() else 0)  #
(GB)
            s = ('%10s' * 2 + '%10.4g' * 6) % (
                '%g/%g' % (epoch, epochs - 1), mem, *mloss, targets.shape[0], imgs.shape[-1])
            pbar.set_description(s)

            # Plot
            if plots and ni < 10:
                f = save_dir / f'train_batch{ni}.jpg'  # filename
                Thread(target=plot_images, args=(imgs, targets, paths, f), daemon=True).start()
                # if tb_writer:
```

```
                #     tb_writer.add_image(f, result, dataformats='HWC', global_step=epoch)
                #     tb_writer.add_graph(torch.jit.trace(model, imgs, strict=False), [])  # add model graph
            elif plots and ni == 10 and wandb_logger.wandb:
                wandb_logger.log({"Mosaics": [wandb_logger.wandb.Image(str(x), caption=x.name) for x
in
                                              save_dir.glob('train*.jpg') if x.exists()]})

        # end batch ------------------------------------------------------------------------------------------------
    # end epoch ----------------------------------------------------------------------------------------------------

    # Scheduler
    lr = [x['lr'] for x in optimizer.param_groups]  # for tensorboard
    scheduler.step()

    # DDP process 0 or single-GPU
    if rank in [-1, 0]:
        # mAP
        ema.update_attr(model, include=['yaml', 'nc', 'hyp', 'gr', 'names', 'stride', 'class_weights'])
        final_epoch = epoch + 1 == epochs
        if not opt.notest or final_epoch:  # Calculate mAP
            wandb_logger.current_epoch = epoch + 1
            results, maps, times = test.test(data_dict,
                                             batch_size=batch_size * 2,
                                             imgsz=imgsz_test,
                                             model=ema.ema,
                                             single_cls=opt.single_cls,
                                             dataloader=testloader,
                                             save_dir=save_dir,
                                             verbose=nc < 50 and final_epoch,
                                             plots=plots and final_epoch,
                                             wandb_logger=wandb_logger,
```

```
                                compute_loss=compute_loss,
                                is_coco=is_coco,
                                v5_metric=opt.v5_metric)

        # Write
        with open(results_file, 'a') as f:
            f.write(s + '%10.4g' * 7 % results + '\n')  # append metrics, val_loss
        if len(opt.name) and opt.bucket:
            os.system('gsutil cp %s gs://%s/results/results%s.txt' % (results_file, opt.bucket, opt.name))

        # Log
        tags = ['train/box_loss', 'train/obj_loss', 'train/cls_loss',  # train loss
                'metrics/precision', 'metrics/recall', 'metrics/mAP_0.5', 'metrics/mAP_0.5:0.95',
                'val/box_loss', 'val/obj_loss', 'val/cls_loss',  # val loss
                'x/lr0', 'x/lr1', 'x/lr2']  # params
        for x, tag in zip(list(mloss[:-1]) + list(results) + lr, tags):
            if tb_writer:
                tb_writer.add_scalar(tag, x, epoch)  # tensorboard
            if wandb_logger.wandb:
                wandb_logger.log({tag: x})  # W&B

        # Update best mAP
        fi = fitness(np.array(results).reshape(1, -1))  # weighted combination of [P, R, mAP@.5, mAP@.5-.95]
        if fi > best_fitness:
            best_fitness = fi
        wandb_logger.end_epoch(best_result=best_fitness == fi)

        # Save model
        if (not opt.nosave) or (final_epoch and not opt.evolve):  # if save
            ckpt = {'epoch': epoch,
```

```
                'best_fitness': best_fitness,
                'training_results': results_file.read_text(),
                'model': deepcopy(model.module if is_parallel(model) else model).half(),
                'ema': deepcopy(ema.ema).half(),
                'updates': ema.updates,
                'optimizer': optimizer.state_dict(),
                'wandb_id': wandb_logger.wandb_run.id if wandb_logger.wandb else None}

            # Save last, best and delete
            torch.save(ckpt, last)
            if best_fitness == fi:
                torch.save(ckpt, best)
            if (best_fitness == fi) and (epoch >= 200):
                torch.save(ckpt, wdir / 'best_{:03d}.pt'.format(epoch))
            if epoch == 0:
                torch.save(ckpt, wdir / 'epoch_{:03d}.pt'.format(epoch))
            elif ((epoch+1) % 25) == 0:
                torch.save(ckpt, wdir / 'epoch_{:03d}.pt'.format(epoch))
            elif epoch >= (epochs-5):
                torch.save(ckpt, wdir / 'epoch_{:03d}.pt'.format(epoch))
            if wandb_logger.wandb:
                if ((epoch + 1) % opt.save_period == 0 and not final_epoch) and opt.save_period != -1:
                    wandb_logger.log_model(
                        last.parent, opt, epoch, fi, best_model=best_fitness == fi)
            del ckpt

    # end epoch ----------------------------------------------------------------------------------------------------
# end training
if rank in [-1, 0]:
    # Plots
    if plots:
```

```
            plot_results(save_dir=save_dir)  # save as results.png
            if wandb_logger.wandb:
                files = ['results.png', 'confusion_matrix.png', *[f'{x}_curve.png' for x in ('F1', 'PR', 'P', 'R')]]
                wandb_logger.log({"Results": [wandb_logger.wandb.Image(str(save_dir / f), caption=f) for f
in files
                                              if (save_dir / f).exists()]})
        # Test best.pt
        logger.info('%g epochs completed in %.3f hours.\n' % (epoch - start_epoch + 1, (time.time() - t0)
/ 3600))
        if opt.data.endswith('coco.yaml') and nc == 80:  # if COCO
            for m in (last, best) if best.exists() else (last):  # speed, mAP tests
                results, _, _ = test.test(opt.data,
                                          batch_size=batch_size * 2,
                                          imgsz=imgsz_test,
                                          conf_thres=0.001,
                                          iou_thres=0.7,
                                          model=attempt_load(m, device).half(),
                                          single_cls=opt.single_cls,
                                          dataloader=testloader,
                                          save_dir=save_dir,
                                          save_json=True,
                                          plots=False,
                                          is_coco=is_coco,
                                          v5_metric=opt.v5_metric)

        # Strip optimizers
        final = best if best.exists() else last  # final model
        for f in last, best:
            if f.exists():
                strip_optimizer(f)  # strip optimizers
        if opt.bucket:
            os.system(f'gsutil cp {final} gs://{opt.bucket}/weights')  # upload
```

```
        if wandb_logger.wandb and not opt.evolve:  # Log the stripped model
            wandb_logger.wandb.log_artifact(str(final), type='model',
                                            name='run_' + wandb_logger.wandb_run.id + '_model',
                                            aliases=['last', 'best', 'stripped'])
        wandb_logger.finish_run()
    else:
        dist.destroy_process_group()
    torch.cuda.empty_cache()
    return results


if __name__ == '__main__':
    parser = argparse.ArgumentParser()
    parser.add_argument('--weights', type=str, default='yolo7.pt', help='initial weights path')
    parser.add_argument('--cfg', type=str, default='', help='model.yaml path')
    parser.add_argument('--data', type=str, default='data/coco.yaml', help='data.yaml path')
    parser.add_argument('--hyp', type=str, default='data/hyp.scratch.p5.yaml', help='hyperparameters path')
    parser.add_argument('--epochs', type=int, default=300)
    parser.add_argument('--batch-size', type=int, default=16, help='total batch size for all GPUs')
    parser.add_argument('--img-size', nargs='+', type=int, default=[640, 640], help='[train, test] image sizes')
    parser.add_argument('--rect', action='store_true', help='rectangular training')
    parser.add_argument('--resume', nargs='?', const=True, default=False, help='resume most recent training')
    parser.add_argument('--nosave', action='store_true', help='only save final checkpoint')
    parser.add_argument('--notest', action='store_true', help='only test final epoch')
    parser.add_argument('--noautoanchor', action='store_true', help='disable autoanchor check')
    parser.add_argument('--evolve', action='store_true', help='evolve hyperparameters')
    parser.add_argument('--bucket', type=str, default='', help='gsutil bucket')
    parser.add_argument('--cache-images', action='store_true', help='cache images for faster training')
```

```
    parser.add_argument('--image-weights', action='store_true', help='use weighted image selection for training')
    parser.add_argument('--device', default='', help='cuda device, i.e. 0 or 0,1,2,3 or cpu')
    parser.add_argument('--multi-scale', action='store_true', help='vary img-size +/- 50%%')
    parser.add_argument('--single-cls', action='store_true', help='train multi-class data as single-class')
    parser.add_argument('--adam', action='store_true', help='use torch.optim.Adam() optimizer')
    parser.add_argument('--sync-bn', action='store_true', help='use SyncBatchNorm, only available in DDP mode')
    parser.add_argument('--local_rank', type=int, default=-1, help='DDP parameter, do not modify')
    parser.add_argument('--workers', type=int, default=8, help='maximum number of dataloader workers')
    parser.add_argument('--project', default='runs/train', help='save to project/name')
    parser.add_argument('--entity', default=None, help='W&B entity')
    parser.add_argument('--name', default='exp', help='save to project/name')
    parser.add_argument('--exist-ok', action='store_true', help='existing project/name ok, do not increment')
    parser.add_argument('--quad', action='store_true', help='quad dataloader')
    parser.add_argument('--linear-lr', action='store_true', help='linear LR')
    parser.add_argument('--label-smoothing', type=float, default=0.0, help='Label smoothing epsilon')
    parser.add_argument('--upload_dataset', action='store_true', help='Upload dataset as W&B artifact table')
    parser.add_argument('--bbox_interval', type=int, default=-1, help='Set bounding-box image logging interval for W&B')
    parser.add_argument('--save_period', type=int, default=-1, help='Log model after every "save_period" epoch')
    parser.add_argument('--artifact_alias', type=str, default="latest", help='version of dataset artifact to be used')
    parser.add_argument('--freeze', nargs='+', type=int, default=[0], help='Freeze layers: backbone of yolov7=50, first3=0 1 2')
    parser.add_argument('--v5-metric', action='store_true', help='assume maximum recall as 1.0 in AP calculation')
    opt = parser.parse_args()


    # Set DDP variables
```

```
    opt.world_size = int(os.environ['WORLD_SIZE']) if 'WORLD_SIZE' in os.environ else 1
    opt.global_rank = int(os.environ['RANK']) if 'RANK' in os.environ else -1
    set_logging(opt.global_rank)
    #if opt.global_rank in [-1, 0]:
    #    check_git_status()
    #    check_requirements()

    # Resume
    wandb_run = check_wandb_resume(opt)
    if opt.resume and not wandb_run:  # resume an interrupted run
        ckpt = opt.resume if isinstance(opt.resume, str) else get_latest_run()  # specified or most recent path
        assert os.path.isfile(ckpt), 'ERROR: --resume checkpoint does not exist'
        apriori = opt.global_rank, opt.local_rank
        with open(Path(ckpt).parent.parent / 'opt.yaml') as f:
            opt = argparse.Namespace(**yaml.load(f, Loader=yaml.SafeLoader))  # replace
        opt.cfg, opt.weights, opt.resume, opt.batch_size, opt.global_rank, opt.local_rank = '', ckpt, True, opt.total_batch_size, *apriori  # reinstate
        logger.info('Resuming training from %s' % ckpt)
    else:
        # opt.hyp = opt.hyp or ('hyp.finetune.yaml' if opt.weights else 'hyp.scratch.yaml')
        opt.data, opt.cfg, opt.hyp = check_file(opt.data), check_file(opt.cfg), check_file(opt.hyp)  # check files
        assert len(opt.cfg) or len(opt.weights), 'either --cfg or --weights must be specified'
        opt.img_size.extend([opt.img_size[-1]] * (2 - len(opt.img_size)))  # extend to 2 sizes (train, test)
        opt.name = 'evolve' if opt.evolve else opt.name
        opt.save_dir = increment_path(Path(opt.project) / opt.name, exist_ok=opt.exist_ok | opt.evolve) # increment run

    # DDP mode
    opt.total_batch_size = opt.batch_size
    device = select_device(opt.device, batch_size=opt.batch_size)
```

```
    if opt.local_rank != -1:
        assert torch.cuda.device_count() > opt.local_rank
        torch.cuda.set_device(opt.local_rank)
        device = torch.device('cuda', opt.local_rank)
        dist.init_process_group(backend='nccl', init_method='env://')  # distributed backend
        assert opt.batch_size % opt.world_size == 0, '--batch-size must be multiple of CUDA device count'
        opt.batch_size = opt.total_batch_size // opt.world_size

    # Hyperparameters
    with open(opt.hyp) as f:
        hyp = yaml.load(f, Loader=yaml.SafeLoader)  # load hyps

    # Train
    logger.info(opt)
    if not opt.evolve:
        tb_writer = None  # init loggers
        if opt.global_rank in [-1, 0]:
            prefix = colorstr('tensorboard: ')
            logger.info(f"{prefix}Start with 'tensorboard --logdir {opt.project}', view at http://localhost:6006/")
            tb_writer = SummaryWriter(opt.save_dir)  # Tensorboard
        train(hyp, opt, device, tb_writer)

    # Evolve hyperparameters (optional)
    else:
        # Hyperparameter evolution metadata (mutation scale 0-1, lower_limit, upper_limit)
        meta = {'lr0': (1, 1e-5, 1e-1),  # initial learning rate (SGD=1E-2, Adam=1E-3)
                'lrf': (1, 0.01, 1.0),  # final OneCycleLR learning rate (lr0 * lrf)
                'momentum': (0.3, 0.6, 0.98),  # SGD momentum/Adam beta1
                'weight_decay': (1, 0.0, 0.001),  # optimizer weight decay
                'warmup_epochs': (1, 0.0, 5.0),  # warmup epochs (fractions ok)
```

```
        'warmup_momentum': (1, 0.0, 0.95),  # warmup initial momentum
        'warmup_bias_lr': (1, 0.0, 0.2),  # warmup initial bias lr
        'box': (1, 0.02, 0.2),  # box loss gain
        'cls': (1, 0.2, 4.0),  # cls loss gain
        'cls_pw': (1, 0.5, 2.0),  # cls BCELoss positive_weight
        'obj': (1, 0.2, 4.0),  # obj loss gain (scale with pixels)
        'obj_pw': (1, 0.5, 2.0),  # obj BCELoss positive_weight
        'iou_t': (0, 0.1, 0.7),  # IoU training threshold
        'anchor_t': (1, 2.0, 8.0),  # anchor-multiple threshold
        'anchors': (2, 2.0, 10.0),  # anchors per output grid (0 to ignore)
        'fl_gamma': (0, 0.0, 2.0),  # focal loss gamma (efficientDet default gamma=1.5)
        'hsv_h': (1, 0.0, 0.1),  # image HSV-Hue augmentation (fraction)
        'hsv_s': (1, 0.0, 0.9),  # image HSV-Saturation augmentation (fraction)
        'hsv_v': (1, 0.0, 0.9),  # image HSV-Value augmentation (fraction)
        'degrees': (1, 0.0, 45.0),  # image rotation (+/- deg)
        'translate': (1, 0.0, 0.9),  # image translation (+/- fraction)
        'scale': (1, 0.0, 0.9),  # image scale (+/- gain)
        'shear': (1, 0.0, 10.0),  # image shear (+/- deg)
        'perspective': (0, 0.0, 0.001),  # image perspective (+/- fraction), range 0-0.001
        'flipud': (1, 0.0, 1.0),  # image flip up-down (probability)
        'fliplr': (0, 0.0, 1.0),  # image flip left-right (probability)
        'mosaic': (1, 0.0, 1.0),  # image mixup (probability)
        'mixup': (1, 0.0, 1.0),   # image mixup (probability)
        'copy_paste': (1, 0.0, 1.0),  # segment copy-paste (probability)
        'paste_in': (1, 0.0, 1.0)}    # segment copy-paste (probability)

with open(opt.hyp, errors='ignore') as f:
    hyp = yaml.safe_load(f)  # load hyps dict
    if 'anchors' not in hyp:  # anchors commented in hyp.yaml
        hyp['anchors'] = 3
```

```
assert opt.local_rank == -1, 'DDP mode not implemented for --evolve'
opt.notest, opt.nosave = True, True  # only test/save final epoch
# ei = [isinstance(x, (int, float)) for x in hyp.values()]  # evolvable indices
yaml_file = Path(opt.save_dir) / 'hyp_evolved.yaml'  # save best result here
if opt.bucket:
   os.system('gsutil cp gs://%s/evolve.txt .' % opt.bucket)  # download evolve.txt if exists

for _ in range(300):  # generations to evolve
   if Path('evolve.txt').exists():  # if evolve.txt exists: select best hyps and mutate
      # Select parent(s)
      parent = 'single'  # parent selection method: 'single' or 'weighted'
      x = np.loadtxt('evolve.txt', ndmin=2)
      n = min(5, len(x))  # number of previous results to consider
      x = x[np.argsort(-fitness(x))][:n]  # top n mutations
      w = fitness(x) - fitness(x).min()  # weights
      if parent == 'single' or len(x) == 1:
         # x = x[random.randint(0, n - 1)]  # random selection
         x = x[random.choices(range(n), weights=w)[0]]  # weighted selection
      elif parent == 'weighted':
         x = (x * w.reshape(n, 1)).sum(0) / w.sum()  # weighted combination

      # Mutate
      mp, s = 0.8, 0.2  # mutation probability, sigma
      npr = np.random
      npr.seed(int(time.time()))
      g = np.array([x[0] for x in meta.values()])  # gains 0-1
      ng = len(meta)
      v = np.ones(ng)
      while all(v == 1):  # mutate until a change occurs (prevent duplicates)
         v = (g * (npr.random(ng) < mp) * npr.randn(ng) * npr.random() * s + 1).clip(0.3, 3.0)
      for i, k in enumerate(hyp.keys()):  # plt.hist(v.ravel(), 300)
```

```
            hyp[k] = float(x[i + 7] * v[i])  # mutate

        # Constrain to limits
        for k, v in meta.items():
            hyp[k] = max(hyp[k], v[1])  # lower limit
            hyp[k] = min(hyp[k], v[2])  # upper limit
            hyp[k] = round(hyp[k], 5)  # significant digits

        # Train mutation
        results = train(hyp.copy(), opt, device)

        # Write mutation results
        print_mutation(hyp.copy(), results, yaml_file, opt.bucket)

    # Plot results
    plot_evolution(yaml_file)
    print(f'Hyperparameter evolution complete. Best results saved as: {yaml_file}\n'
          f'Command to train a new model with these hyperparameters: $ python train.py --hyp {yaml_file}')
```

appendix b Coco format

```
 from detectron2.data.datasets import register_coco_instances
# register_coco_instances("my_dataset_train", {}, "json_annotation_train.json", "path/to/image/dir")
# register_coco_instances("my_dataset_val", {}, "json_annotation_val.json", "path/to/image/dir")
```

Apendix c running Detectron

```
!python -m pip install pyyaml==5.1
import sys, os, distutils.core
# Note: This is a faster way to install detectron2 in Colab, but it does not include all functionalities.
# See https://detectron2.readthedocs.io/tutorials/install.html for full installation instructions
!git clone 'https://github.com/facebookresearch/detectron2'
```

```
dist = distutils.core.run_setup("./detectron2/setup.py")
!python -
m pip install {' '.join([f"'{x}'" for x in dist.install_requires])}
sys.path.insert(0, os.path.abspath('./detectron2'))

# Properly install detectron2. (Please do not install twice in both way
s)
# !python -
m pip install 'git+https://github.com/facebookresearch/detectron2.git'
import torch, detectron2
!nvcc --version
TORCH_VERSION = ".".join(torch.__version__.split(".")[:2])
CUDA_VERSION = torch.__version__.split("+")[-1]
print("torch: ", TORCH_VERSION, "; cuda: ", CUDA_VERSION)
print("detectron2:", detectron2.__version__)
# Some basic setup:
# Setup detectron2 logger
import detectron2
from detectron2.utils.logger import setup_logger
setup_logger()

# import some common libraries
import numpy as np
import os, json, cv2, random
from google.colab.patches import cv2_imshow

# import some common detectron2 utilities
from detectron2 import model_zoo
from detectron2.engine import DefaultPredictor
from detectron2.config import get_cfg
from detectron2.utils.visualizer import Visualizer
from detectron2.data import MetadataCatalog, DatasetCatalog
dataset_dicts = get_balloon_dicts("balloon/train")
for d in random.sample(dataset_dicts, 3):
    img = cv2.imread(d["file_name"])
    visualizer = Visualizer(img[:, :, ::-
1], metadata=balloon_metadata, scale=0.5)
    out = visualizer.draw_dataset_dict(d)
    cv2_imshow(out.get_image()[:, :, ::-1])
rom detectron2.engine import DefaultTrainer

cfg = get_cfg()
cfg.merge_from_file(model_zoo.get_config_file("COCO-
InstanceSegmentation/mask_rcnn_R_50_FPN_3x.yaml"))
cfg.DATASETS.TRAIN = ("balloon_train",)
cfg.DATASETS.TEST = ()
cfg.DATALOADER.NUM_WORKERS = 2
```

```
cfg.MODEL.WEIGHTS = model_zoo.get_checkpoint_url("COCO-
InstanceSegmentation/mask_rcnn_R_50_FPN_3x.yaml")  # Let training initi
alize from model zoo
cfg.SOLVER.IMS_PER_BATCH = 2  # This is the real "batch size" commonly
known to deep learning people
cfg.SOLVER.BASE_LR = 0.00025  # pick a good LR
cfg.SOLVER.MAX_ITER = 300    # 300 iterations seems good enough for thi
s toy dataset; you will need to train longer for a practical dataset
cfg.SOLVER.STEPS = []        # do not decay learning rate
cfg.MODEL.ROI_HEADS.BATCH_SIZE_PER_IMAGE = 128   # The "RoIHead batch s
ize". 128 is faster, and good enough for this toy dataset (default: 512
)
cfg.MODEL.ROI_HEADS.NUM_CLASSES = 1  # only has one class (ballon). (se
e https://detectron2.readthedocs.io/tutorials/datasets.html#update-the-
config-for-new-datasets)
# NOTE: this config means the number of classes, but a few popular unof
ficial tutorials incorrect uses num_classes+1 here.

os.makedirs(cfg.OUTPUT_DIR, exist_ok=True)
trainer = DefaultTrainer(cfg)
trainer.resume_or_load(resume=False)
trainer.train()
%load_ext tensorboard
%tensorboard --logdir output
```