ज्ञानेन प्रकाशते जगत्
INDUS UNIVERSITY

NAAC ACCREDITED

OBJECTIFY

REAL-TIME OBJECT DETECTION

USING **DEEP LEARNING** WITH **PYTHON** & **PYTORCH**

ज्ञानेन प्रकाशते जगत्
INDUS UNIVERSITY

NAAC ACCREDITED

PREPARED BY

Olwin Christian *IU2141230136*
Deepak Soni *IU2141230282*

Department of Computer Science & Engineering, I.I.T.E, Indus University, Ahmedabad

INTERNAL GUIDE

Dr. Sheetal Pandya
Assistant Professor

Department of Computer Science & Engineering, I.I.T.E, Indus University, Ahmedabad

PROJECT OVERVIEW

The **Objectify** project develops an **advanced** model to **analyze** football video clips, **aiming** to identify and track *players*, the *ball*, and *referees* in real time. Using **deep learning** frameworks like **Faster R-CNN** and **YOLOv8**, the project **focuses** on efficiently **processing** video frames to **deliver** precise object detection with **bounding boxes** and **class labels**.

Key elements include **model training** and **testing** on custom-annotated datasets in Google Colab using tools like **PyTorch** and **Roboflow**.

LITERATURE REVIEW

The **Objectify** project builds upon key research advancements in **deep learning** and **computer vision** for object detection.

- **Deep Learning in Object Detection**: **Faster R-CNN**, introduced by **Shaoqing Ren et al.**, uses **Region Proposal Networks** integrated with **CNNs** to enhance both **speed** and **accuracy**, establishing a **benchmark** in object detection.

- **YOLO** (You Only Look Once): **Joseph Redmon et al.'s** YOLO framework changed real-time detection by **evaluating** images in a **single** step, achieving **faster** detection rates. **YOLOv8**, the **latest** version, improves **performance** for dynamic **settings** like sports.

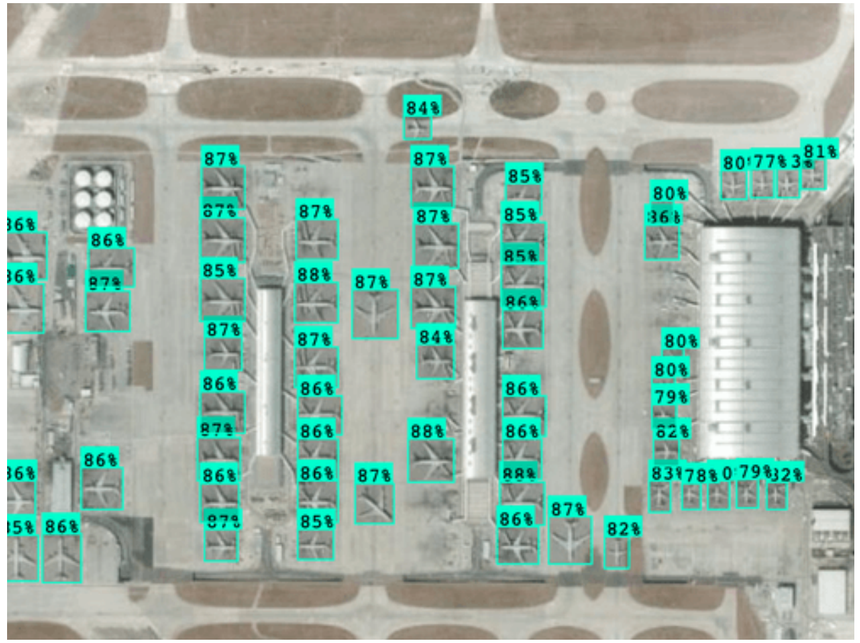
- **Transfer Learning for Object Detection**: Research on transfer learning highlights its **efficiency** in training models on **limited annotated datasets**. Fine-tuning **pre-trained** models saves **time** and maintains high **accuracy** for specific detection **tasks**.

In **summary**, the **Objectify** project leverages these **advancements** to develop a real-time detection model **optimized** for football video analysis, guiding effective **application** of object detection in live, dynamic environments.

APPLICATIONS

The object detection model **YOLOv8**, developed in this project has several **practical** and **impactful** applications across various **industries** including, **Autonomous Smart Solutions.**

Autonomous Inventory Detection



Autonomous Airfield Surveillance

APPLICATIONS

The object detection model developed in this project has several practical and impactful applications across various industries including implementing, **Smart City.**

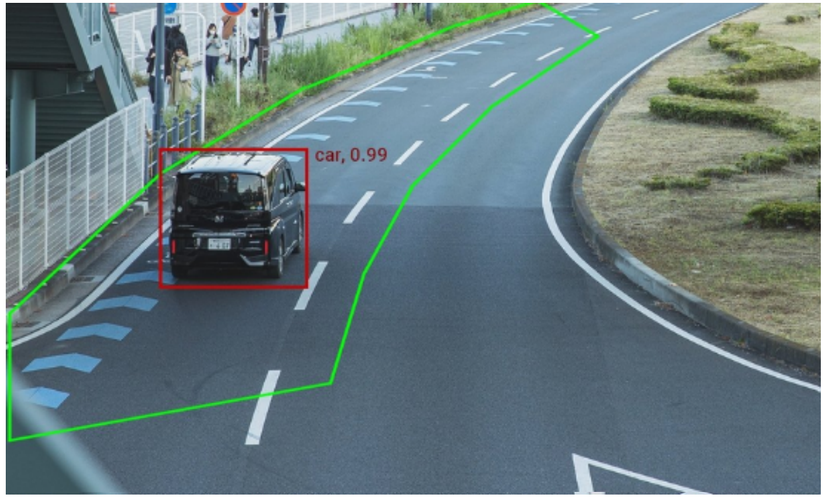
Pothole Detection

Waste Management

APPLICATIONS

The object detection model developed in this project has several practical and impactful applications across various industries including, **Autonomous Detection and Identification.**

License Plate Detection



Anomaly Vehicle Detection

COMPONENTS

Roboflow

Roboflow is used for **dataset** management, streamlining the **annotation** and **preparation** of the **Football** dataset for this project. It allows us to efficiently **annotate** images, **create** training and **validation** splits, and generate **customized** datasets, **improving** the training process for our object detection model.

Netron

Netron is a powerful **visualization** tool that enables us to **inspect** and **analyze** the **architecture** of our trained models. It provides a user-friendly **interface** to view model **layers**, **shapes,** and **parameters**, helping us understand the model's **structure** and **performance**. This insight is crucial for **fine-tuning** and **optimizing** our object detection algorithms.

TensorBoard

TensorBoard is a **visualization** tool that assists in **monitoring** and **analyzing** the training process. It provides insights into model performance **metrics**, **loss graphs**, and **training** progress, making it easier to **diagnose** issues and **optimize** training strategies.

COMPONENTS

Python Programming

The **core** of the project is developed using **Python**, a versatile programming language widely used in **machine learning** and **deep learning** projects. Python facilitates **model** training, **data** handling, and **deploying** object detection models.

PyTorch Framework

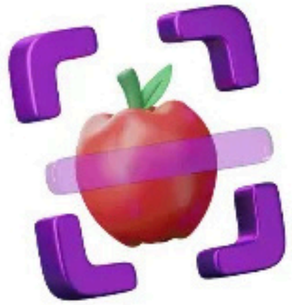
The project utilizes **PyTorch**, an open-source deep learning framework that allows for **flexibility** and **speed** in model building and training. PyTorch is **used** for developing, **training**, and **deploying** deep learning models.

Convolutional Neural Networks (CNNs)

CNNs are the **backbone** of the project's object detection process. These networks **automatically** learn **spatial hierarchies** of features through **back propagation**, which is crucial for **identifying** objects in **images** and **videos.**

Some Practical Applications of CNN

Image classification

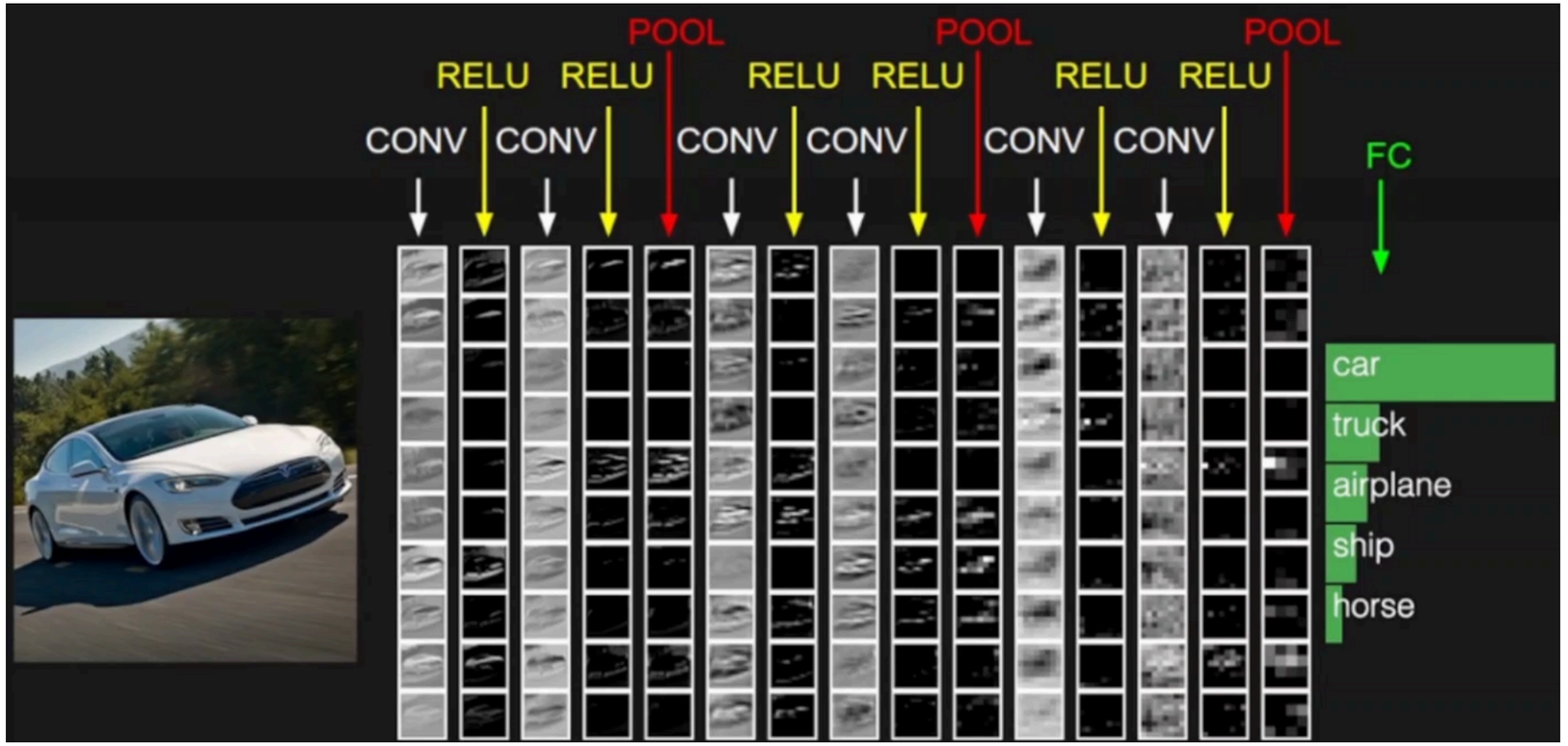


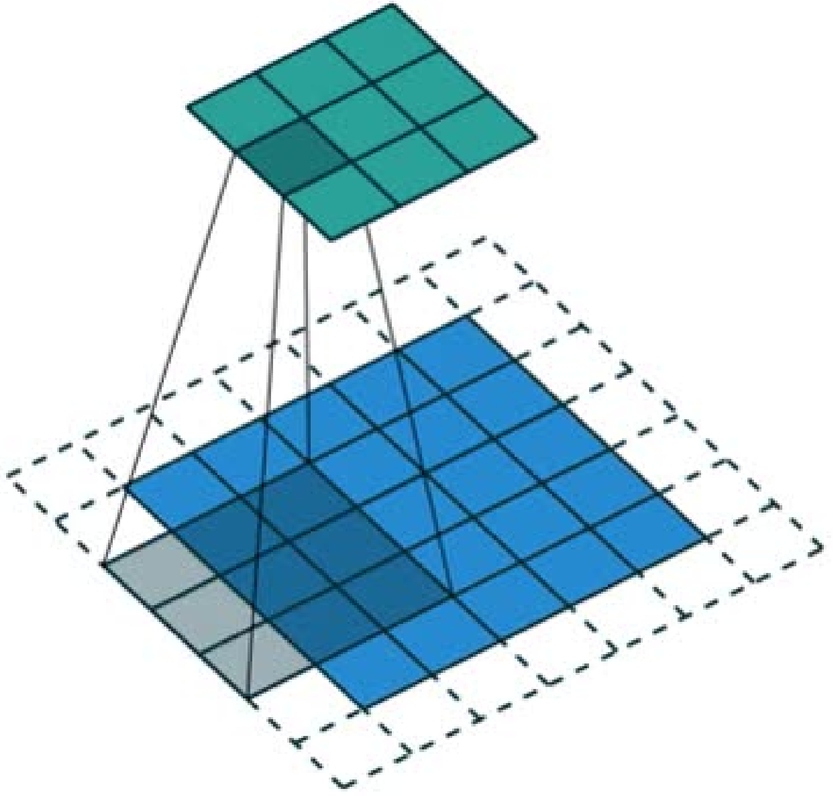
Object detection

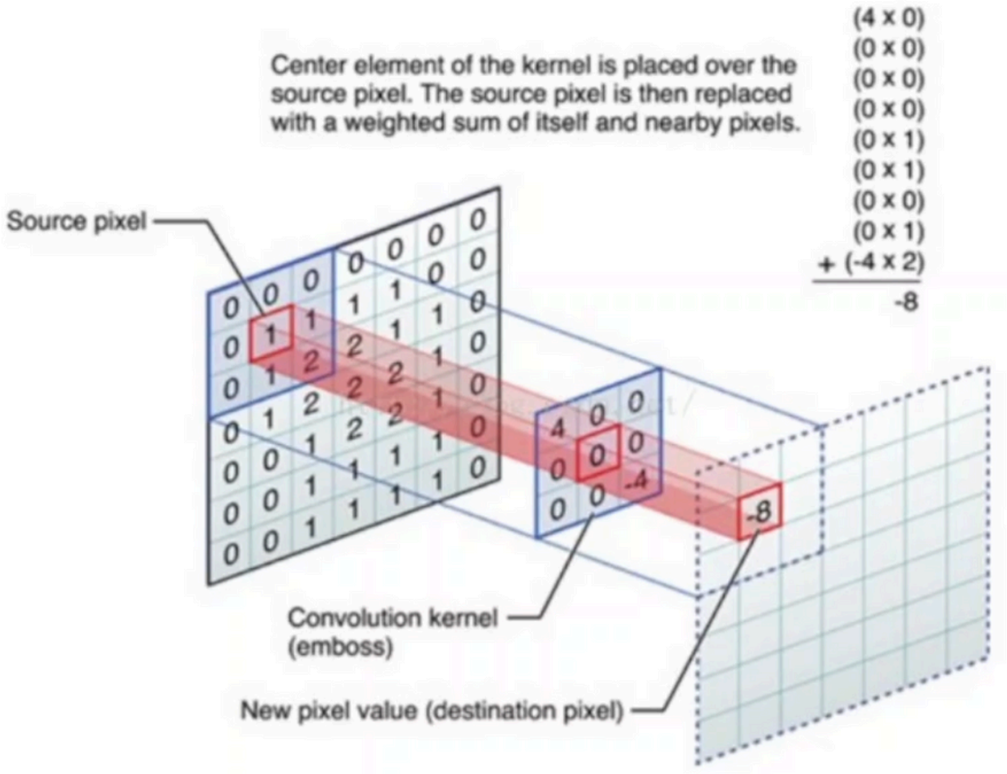
Facial recognition

CNN ARCHITECTURE

- CONV: Convolutional kernel layer
- RELU: Activation function layer
- POOL: Pooling layer for dimension reduction
- FC: Fully connection layer

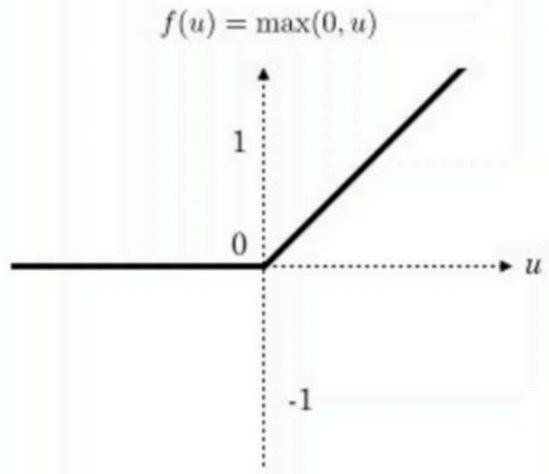




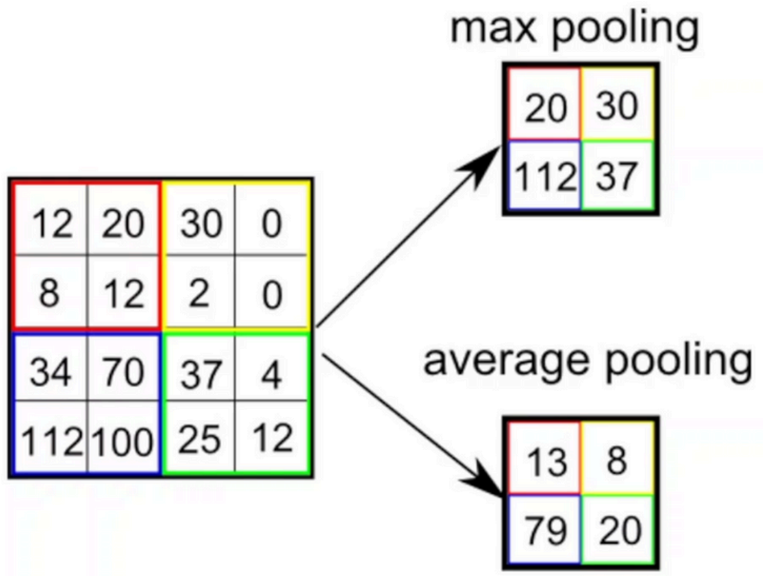

Center element of the kernel is placed over the source pixel. The source pixel is then replaced with a weighted sum of itself and nearby pixels.
(4 x 0)
(0 x 0)
(0 x 0)
(0 x 0)
(0 x 1)
(0 x 1)
(0 x 0)
(0 x 1)
+ (-4 x 2)
-8
Source pixel
Convolution kernel (emboss)
New pixel value (destination pixel)
-8

Activation Layer (ReLU)

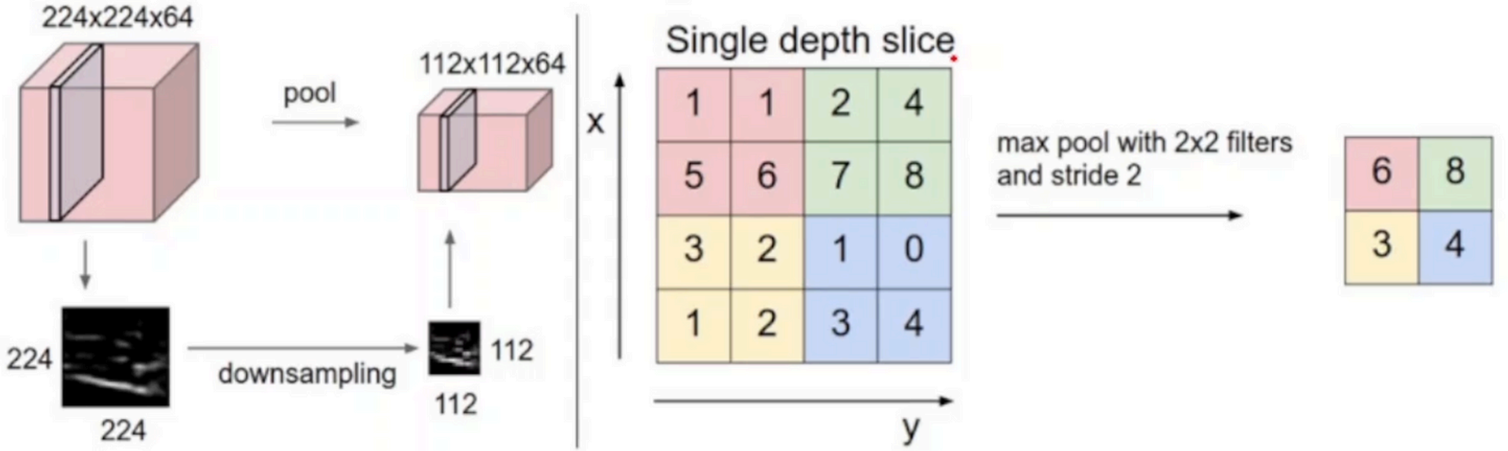
rectified linear function, $f(x) = max(0, x)$



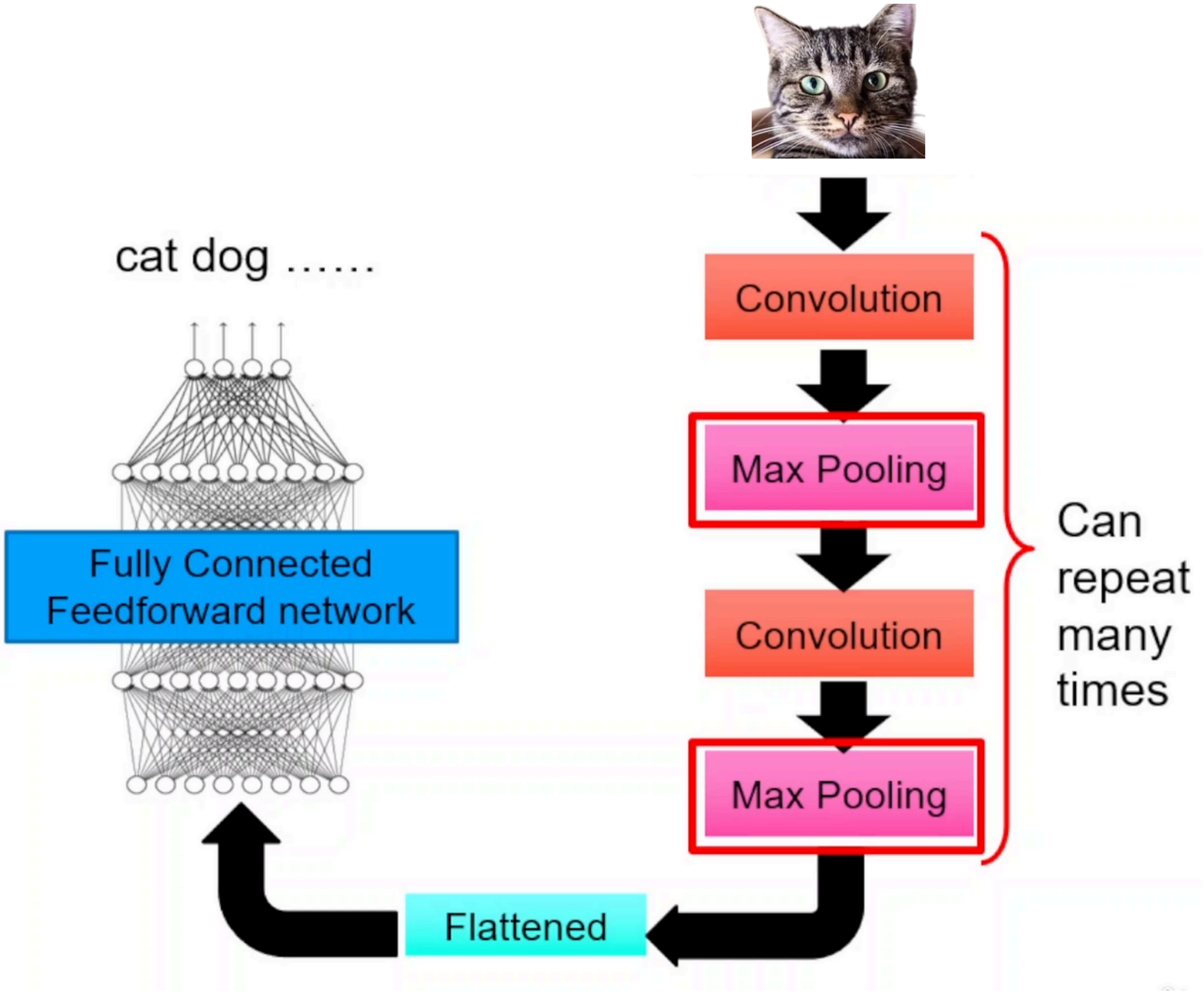
Pooling



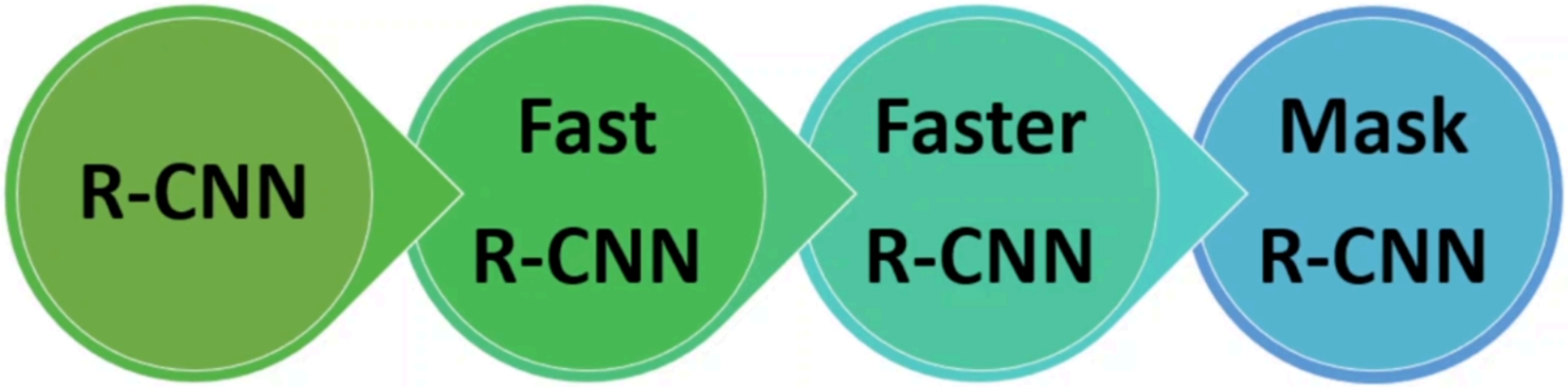
Pooling layer



Pooling layer downsamples the volume spatially, independently in each depth slice of the input volume. **Left:** In this example, the input volume of size [224x224x64] is pooled with filter size 2, stride 2 into output volume of size [112x112x64]. Notice that the volume depth is preserved. **Right:** The most common downsampling operation is max, giving rise to **max pooling**, here shown with a stride of 2. That is, each max is taken over 4 numbers (little 2x2 square).

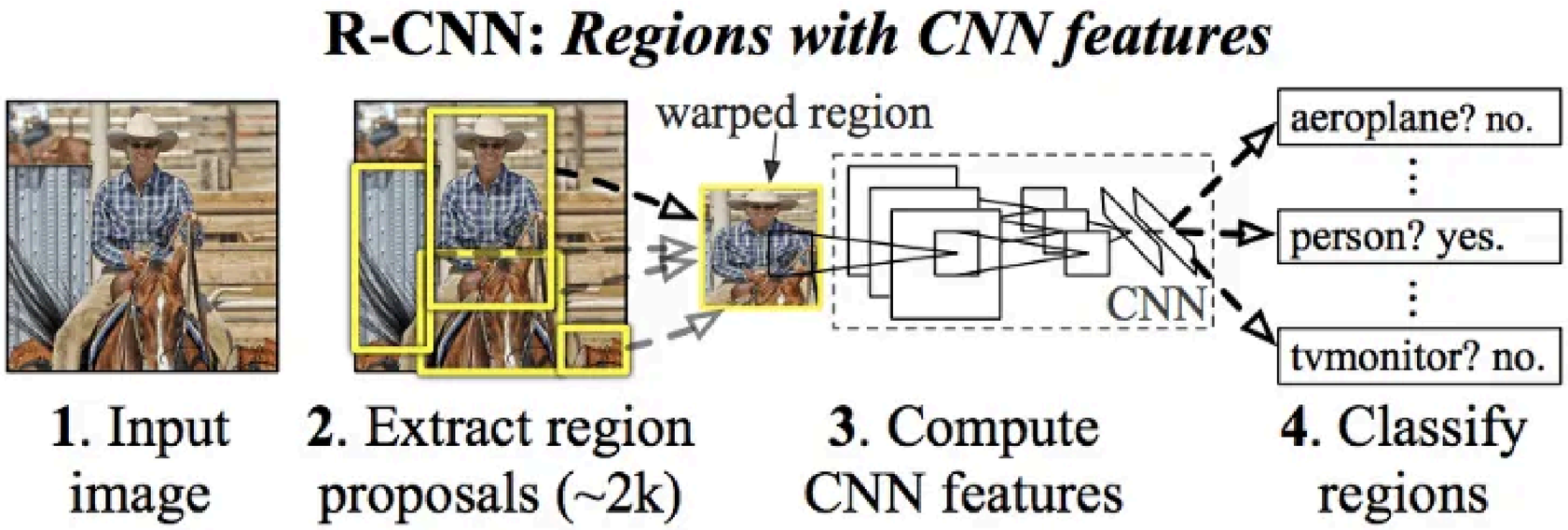

cat dog ……
Fully Connected Feedforward network
Flattened
Convolution
Max Pooling
Convolution
Max Pooling
Can repeat many times

R-CNN FAMILY



R-CNN

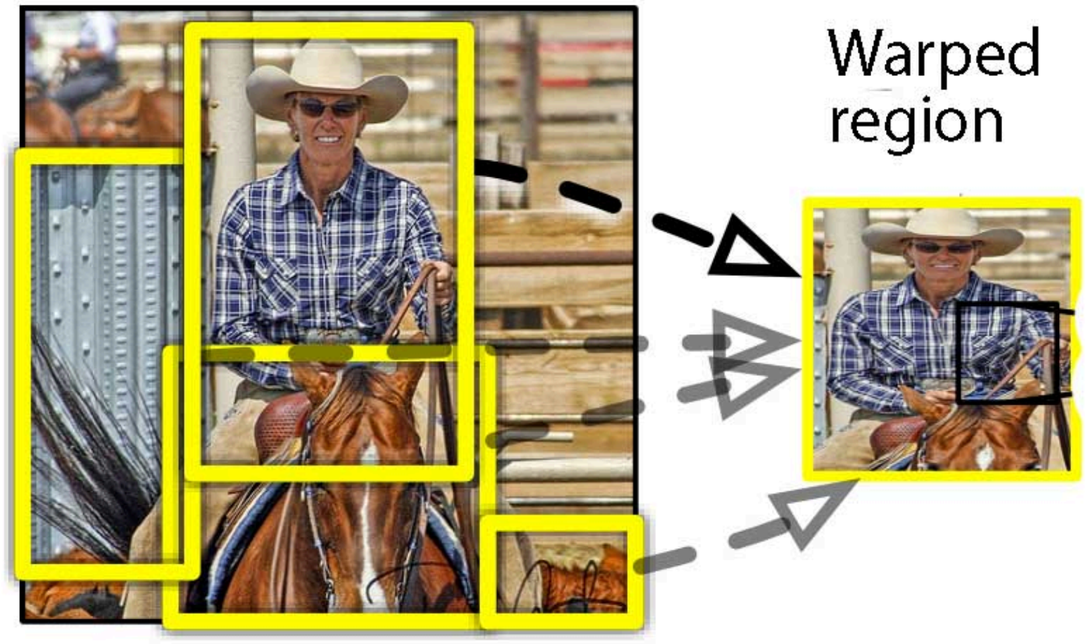
Uses **Selective Search** to propose **regions**, then **classifies** each region using a **CNN**.



For an **image** with **multiple** objects, it **proposes** regions and **classifies** them as *aeroplane*, *person*, *or tvmonitor*.

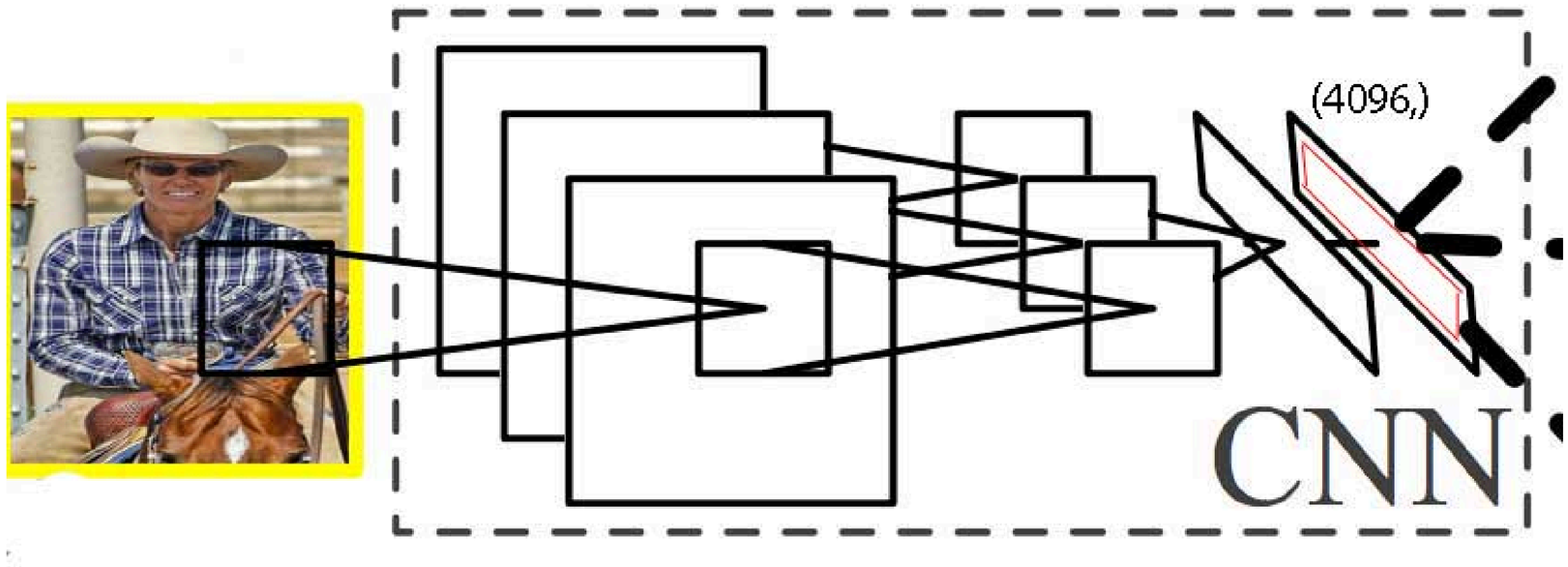
R-CNN – WARPED REGION

*In R-CNN, after **selecting** regions of interest (RoIs), each region is **resized** to a **fixed** size of 227 x 227 x 3 before being processed by a **Convolutional Neural Network** (CNN). This resizing ensures **consistent** input dimensions for **accurate** object extraction from the **identified** areas. This process is known as, **Warping.***



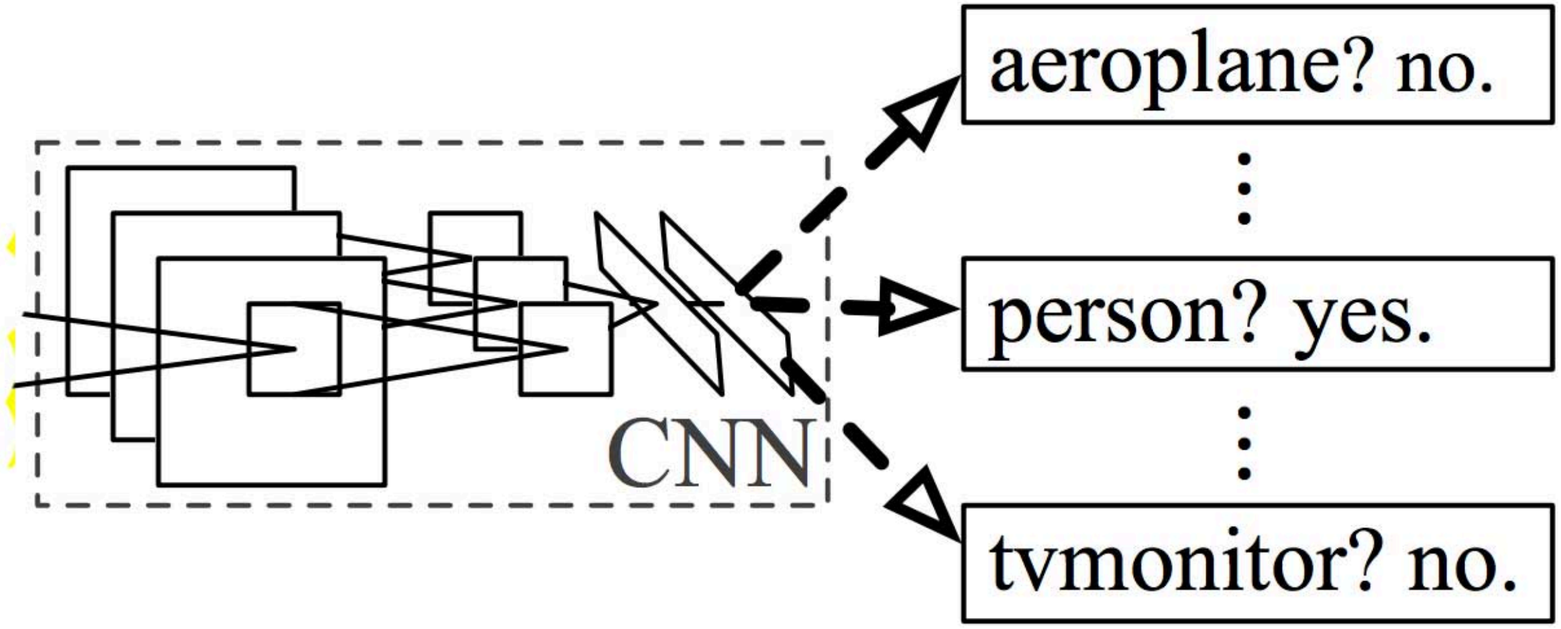
R-CNN – FEATURE EXTRACTION

*A **wrapped** input for CNN will be processed to extract the object of size **4096** dimensions.*

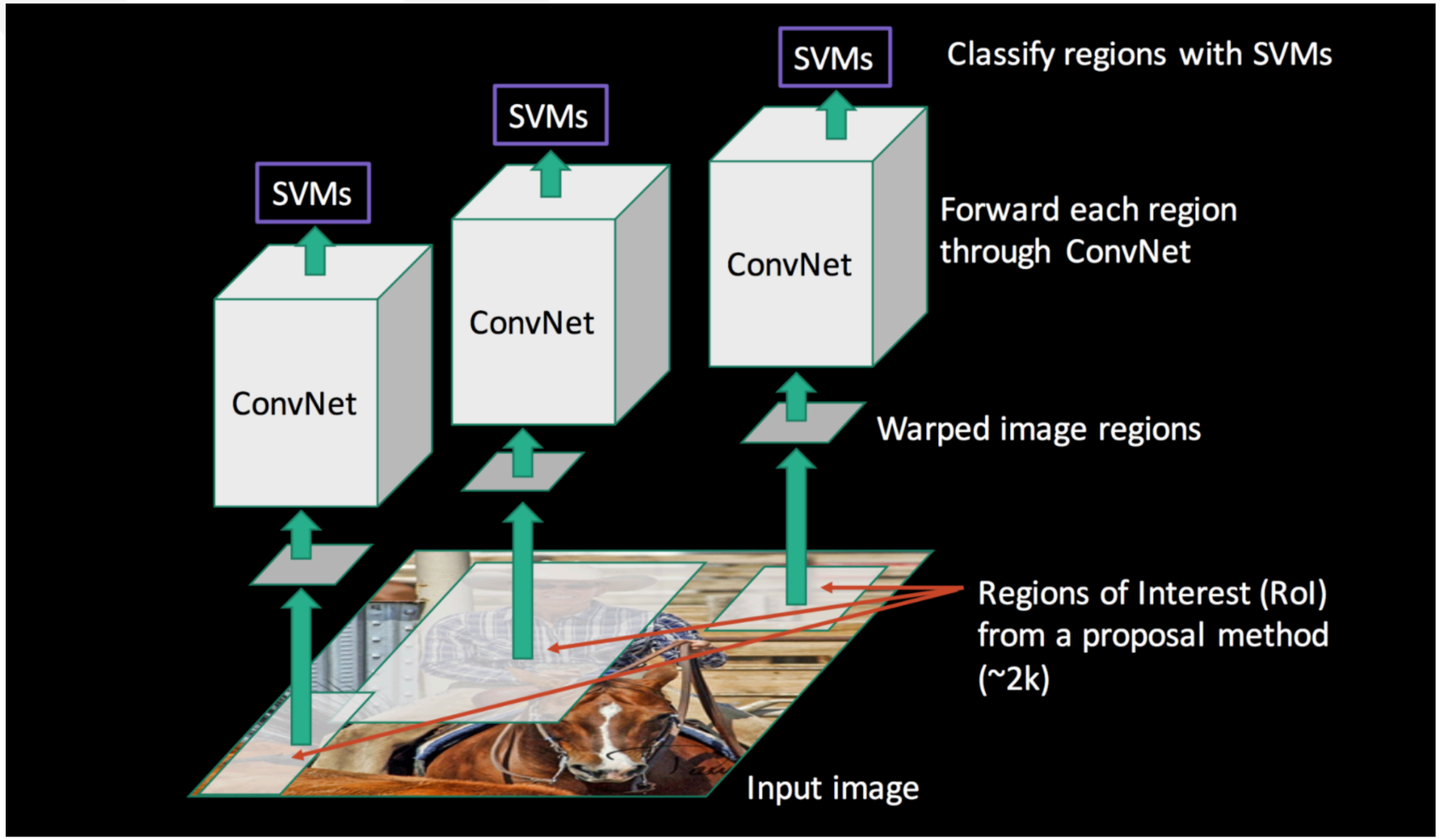


R-CNN – CLASSIFICATION

*The basic R-CNN consists of an **SVM** classifier to **segregate** different objects into their class.*



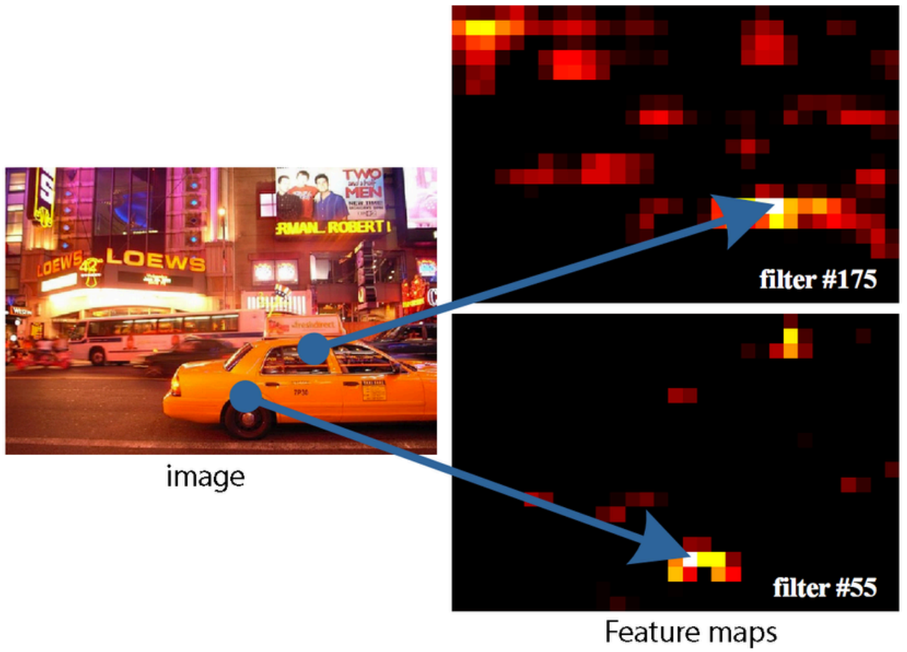
R-CNN – ARCHITECTURE



FAST R-CNN

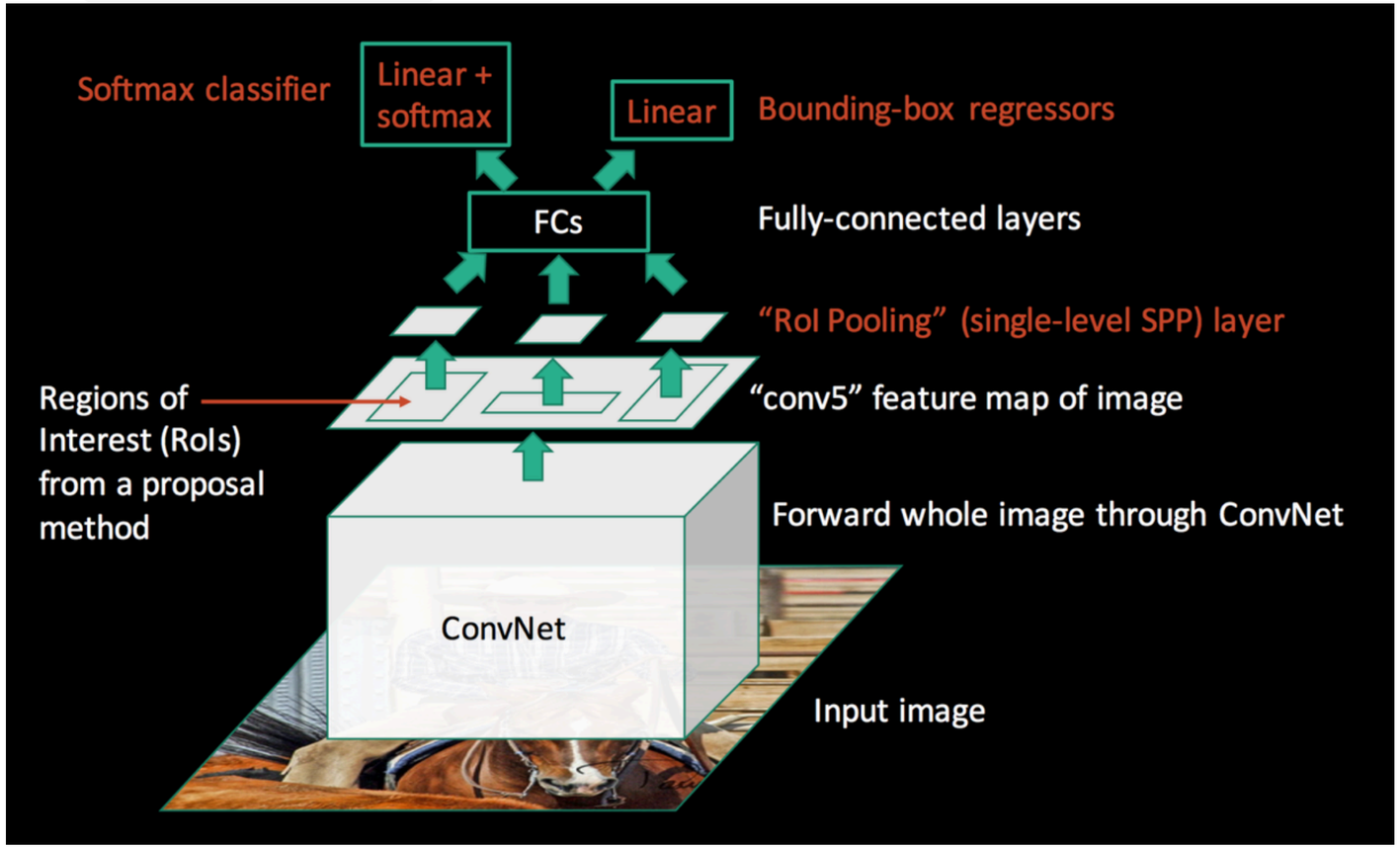
In **Fast R-CNN** instead of performing **maximum pooling**, we perform **ROI pooling** for **utilising** a **Single Feature Map** for all the regions. This **warps** ROIs into **one single layer**; the ROI pooling layer uses **max pooling** to **convert** the features.

*We can clearly see the effects of **Maximum Pooling** layer black colour is more **highlighted** than other light colours which also helps in **finding** the dark coloured objects in the image.*



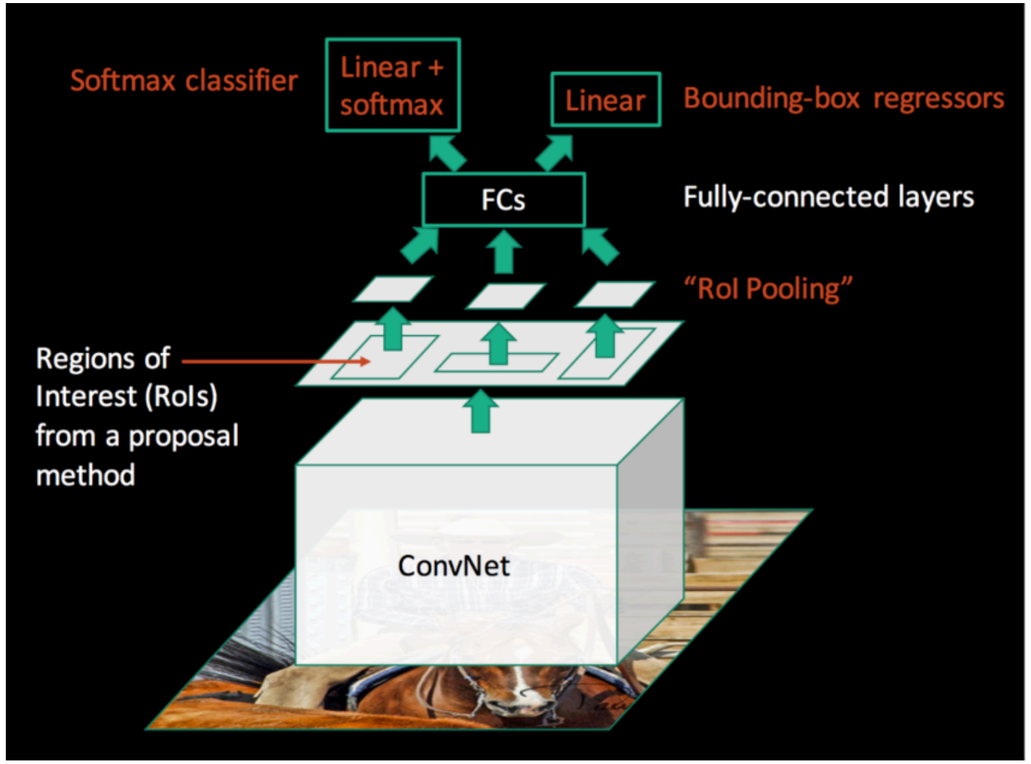
FAST R-CNN

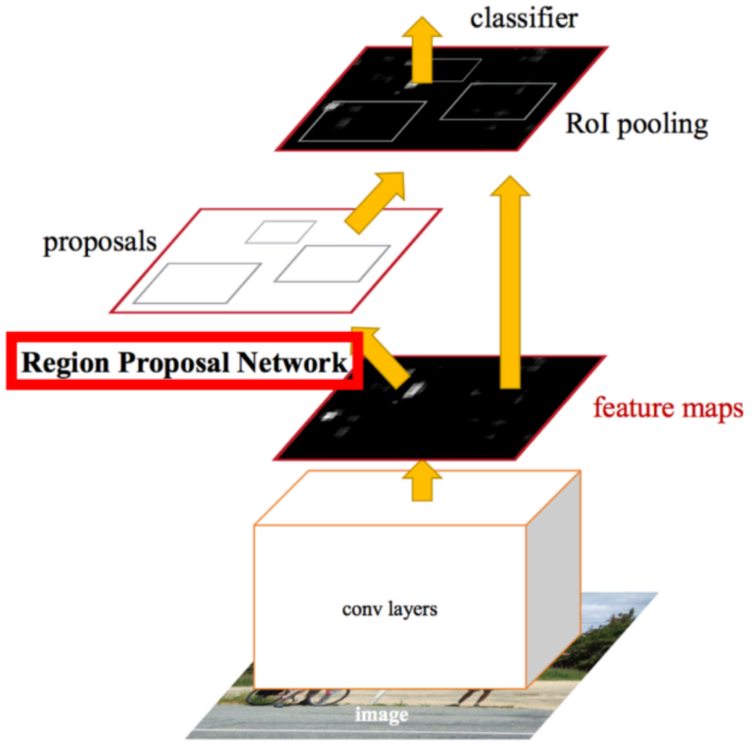
Since **Max Pooling** is also working here, that's why we can consider **Fast R-CNN** as an **upgrade** of the **SPPNet**. Instead of **generating** layers in a **pyramid** shape, it **generates** only **one layer**.



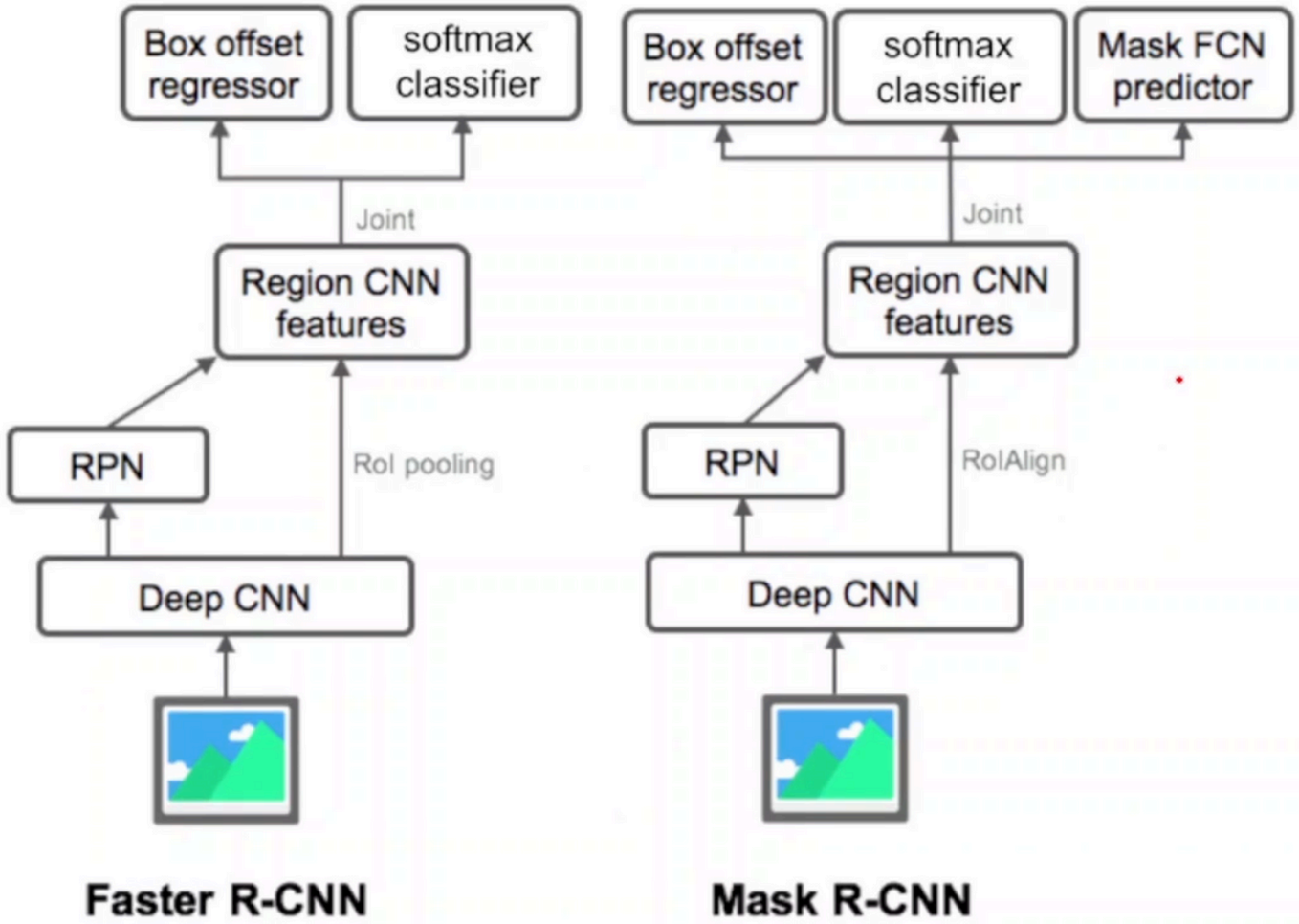
FASTER R-CNN

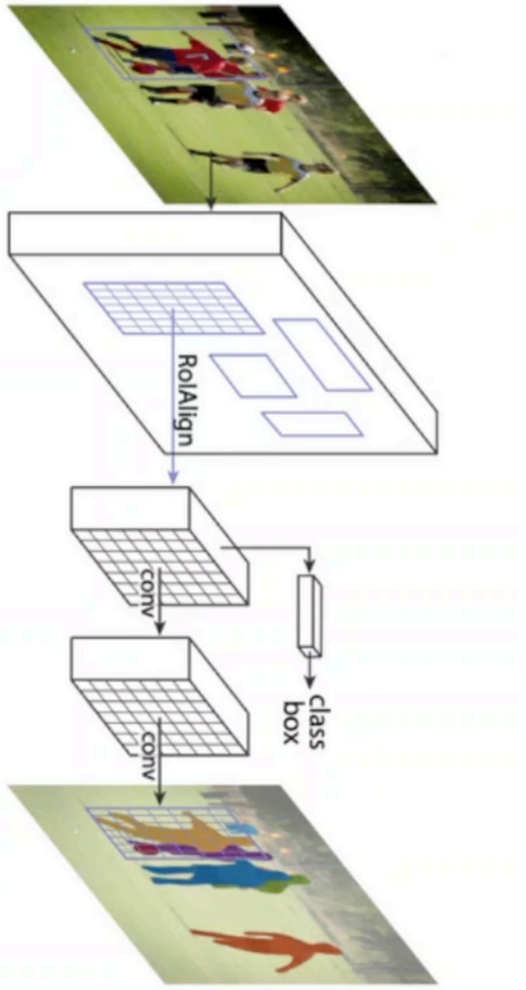
So far, we've noted that **SPPNet** and **Fast R-CNN lack** methods for selecting **regions of interest**, which **distinguishes** Fast R-CNN from Faster R-CNN. **Faster R-CNN** introduces a **Region Proposal Network** (RPN) that generates **region proposals**. The RPN takes the **feature map** as input during training and outputs these proposals, which are then sent to the **ROI pooling layer** for further processing.

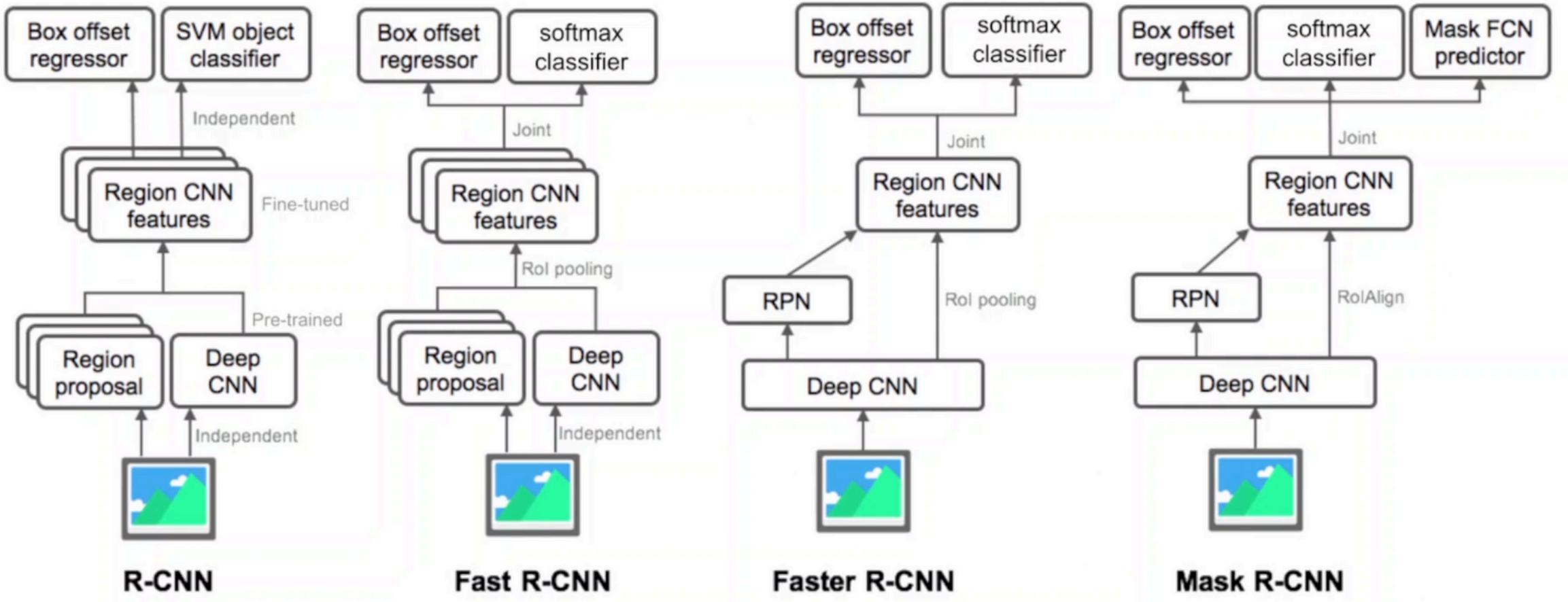




MASK R-CNN


Box offset regressor
softmax classifier
Joint
Region CNN features
RPN
RoI pooling
Deep CNN
Faster R-CNN
Box offset regressor
softmax classifier
Mask FCN predictor
Joint
Region CNN features
RPN
RoIAlign
Deep CNN
Mask R-CNN


RoIAlign
conv
conv
class
box


Box offset regressor
SVM object classifier
Independent
Region CNN features
Fine-tuned
Pre-trained
Region proposal
Deep CNN
Independent
R-CNN
Box offset regressor
softmax classifier
Joint
Region CNN features
RoI pooling
Region proposal
Deep CNN
Independent
Fast R-CNN
Box offset regressor
softmax classifier
Joint
Region CNN features
RPN
RoI pooling
Deep CNN
Faster R-CNN
Box offset regressor
softmax classifier
Mask FCN predictor
Joint
Region CNN features
RPN
RoIAlign
Deep CNN
Mask R-CNN

YOLO

YOLO (You Only Look Once) has gained immense **popularity** in the field of object detection due to several key factors.

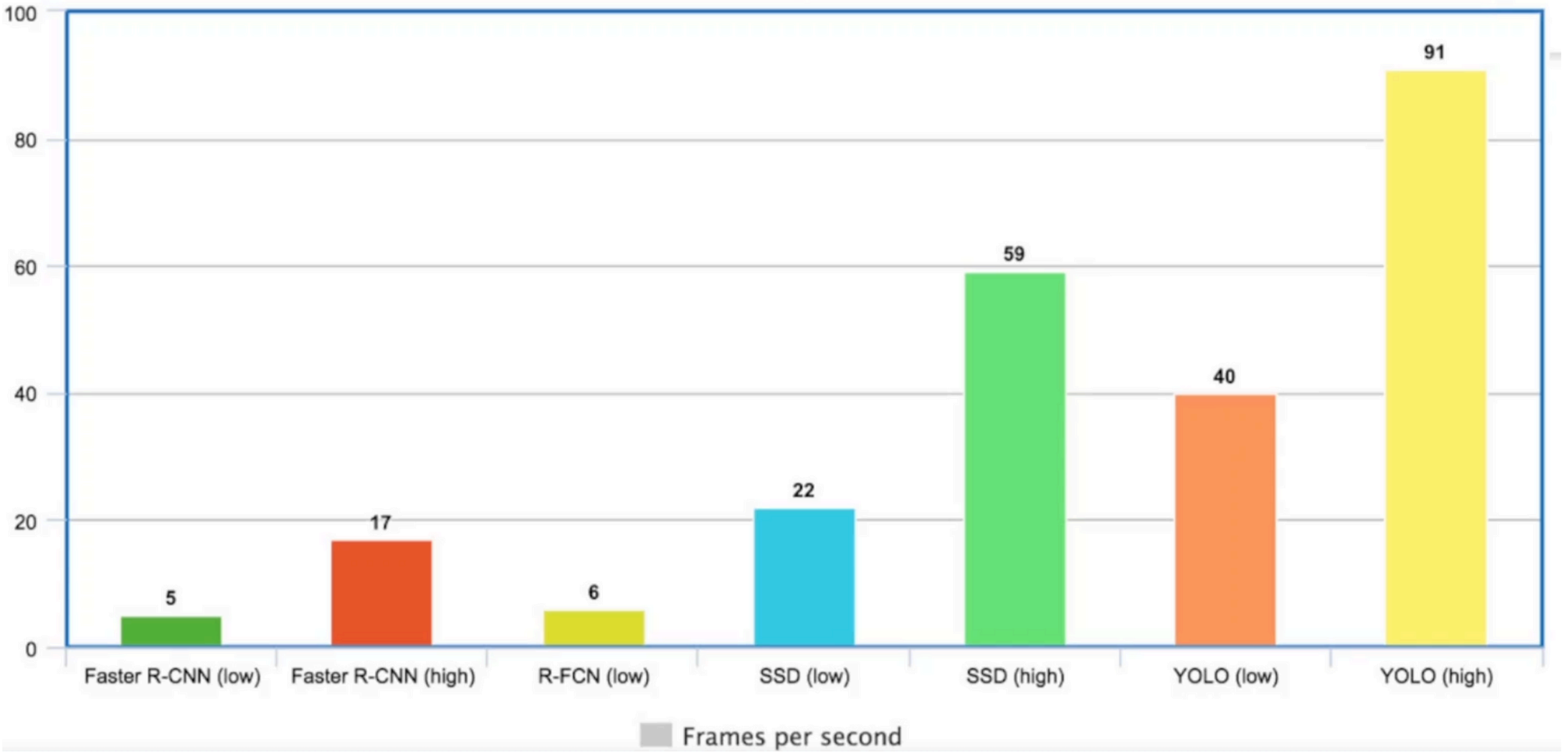
- **Speed**: YOLO is **renowned** for its real-time **processing** capabilities, allowing it to detect objects **quickly** by predicting **bounding boxes** and **class probabilities** from a *single forward pass* of the network.

- **Detection Accuracy**: Despite its **speed**, YOLO **maintains** high detection accuracy, **effectively** identifying objects in **various** conditions, which is crucial for **applications** requiring **reliable** performance.

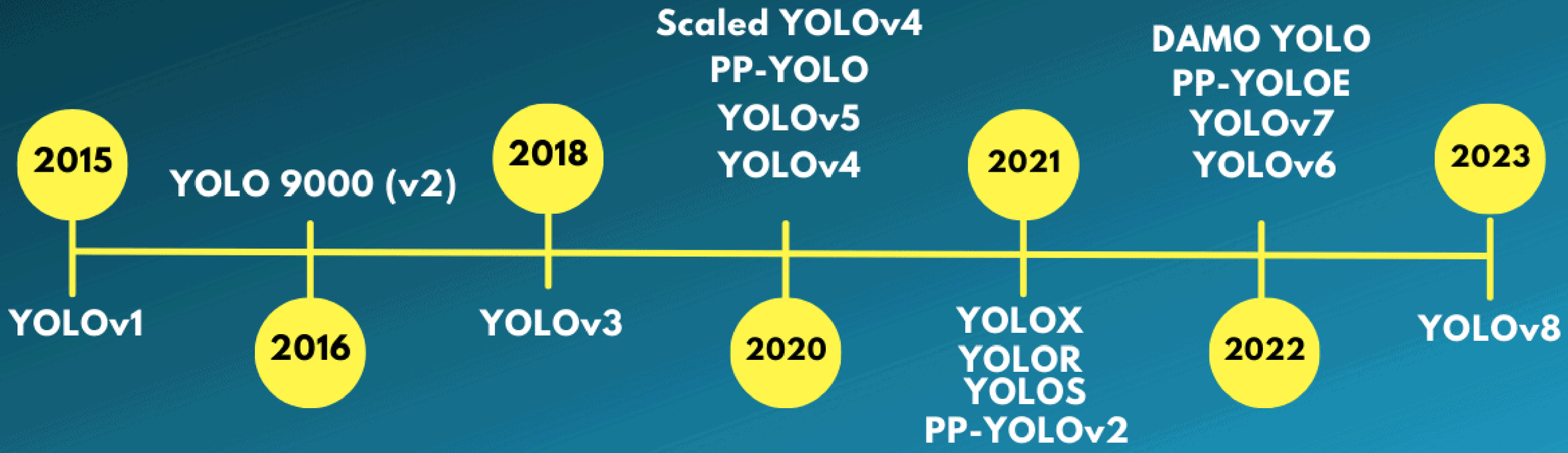
- **Good Generalization**: YOLO **exhibits** strong **generalization** capabilities, enabling it to **perform** well across different datasets and object classes without **extensive** retraining.

- **Open Source**: Being an open-source framework allows developers and researchers to **access** the model easily, **modify** it for specific **applications**, and contribute to its **continuous** improvement, **fostering** a vibrant community around it.

These **features** collectively make YOLO a **leading** choice for object detection tasks across various **domains**.


100
80
60
40
20
0
5
17
6
22
59
40
91
Faster R-CNN (low)
Faster R-CNN (high)
R-FCN (low)
SSD (low)
SSD (high)
YOLO (low)
YOLO (high)
Frames per second

YOLO Object Detection Models Timeline



YOLOV8

YOLOv8, developed by **Ultralytics**, is the latest **advancement** in the YOLO series, supporting object detection, **image classification,** and **instance segmentation**. Building on **YOLOv5**'s legacy, **YOLOv8** introduces **enhanced architecture** and **a user-friendly** experience.

- **Versatile Capabilities**
  Supports object detection, image classification, and instance segmentation.

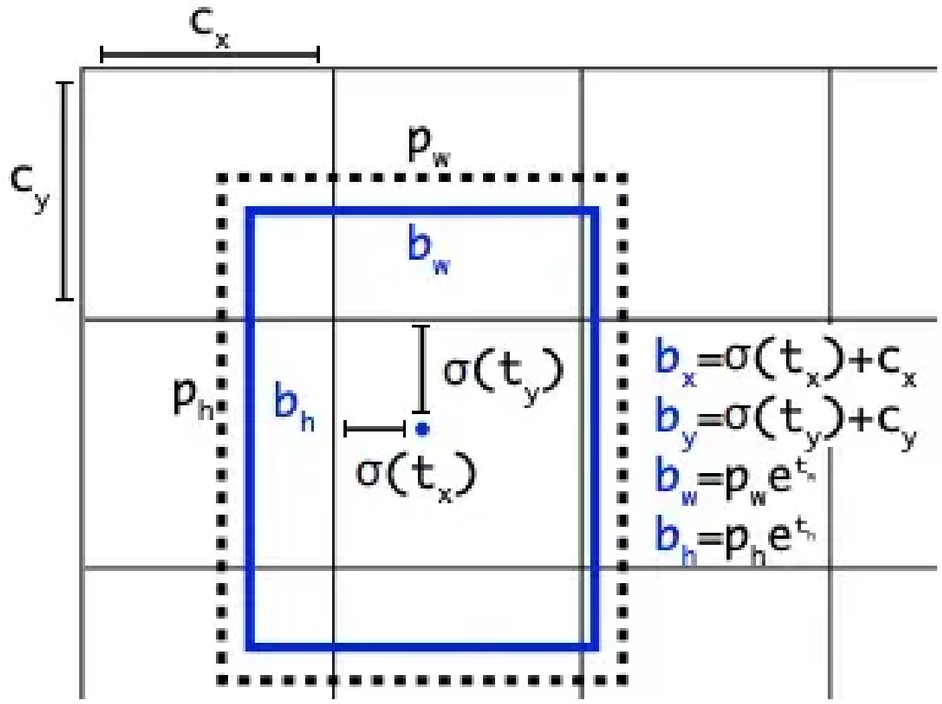
- **Developed by Ultralytics**
  Creators of the groundbreaking **YOLOv5**, now with enhanced architecture and user experience.

- **Community-Driven and Actively Supported**
  Ongoing improvements, new features, and long-term support in collaboration with the community.

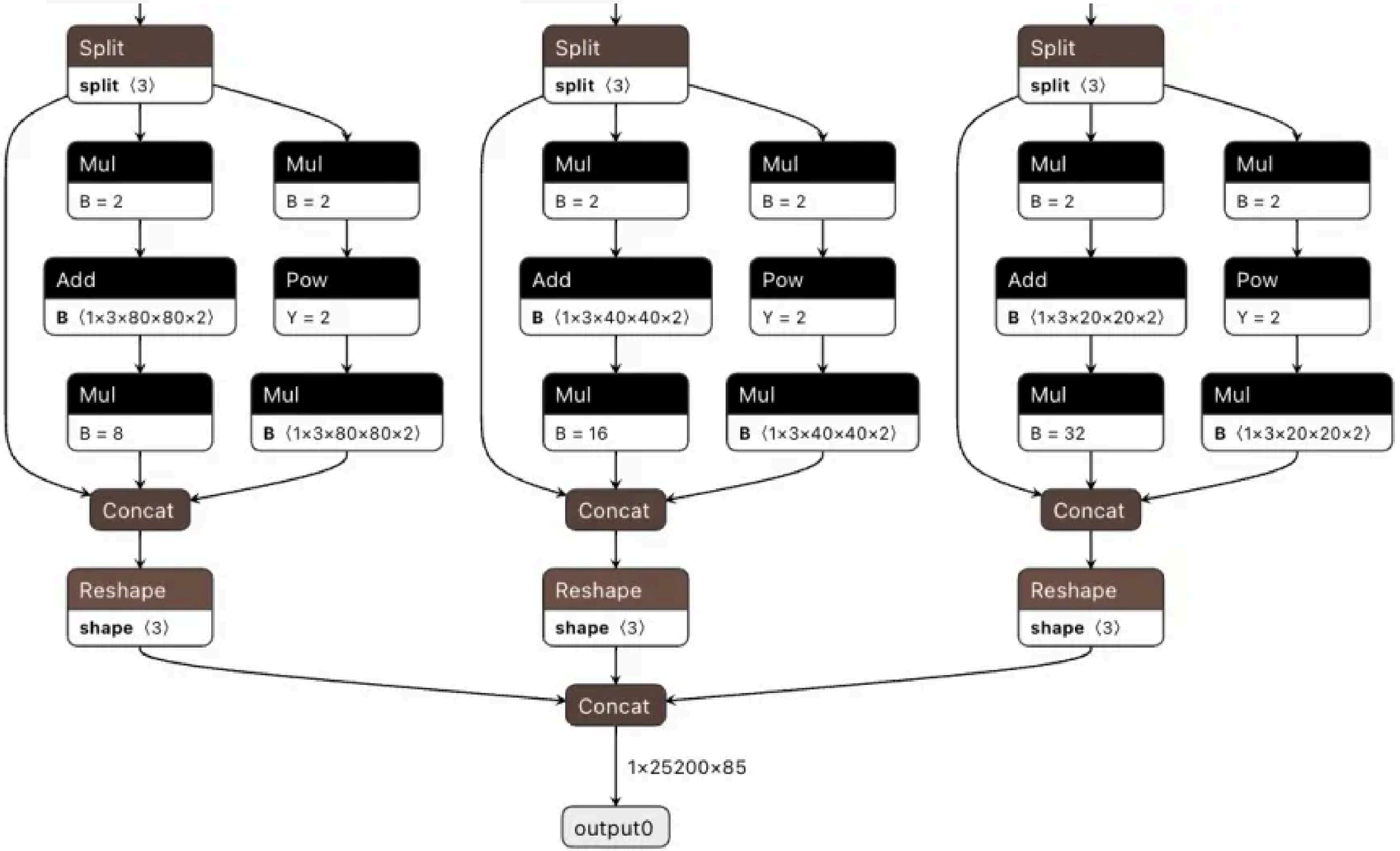
In summary, YOLOv8 excels in object detection and segmentation, combining advanced capabilities, intuitive design, and active community engagement, making it a powerful tool for diverse applications.

YOLOV8 – ANCHOR FREE DETECTION

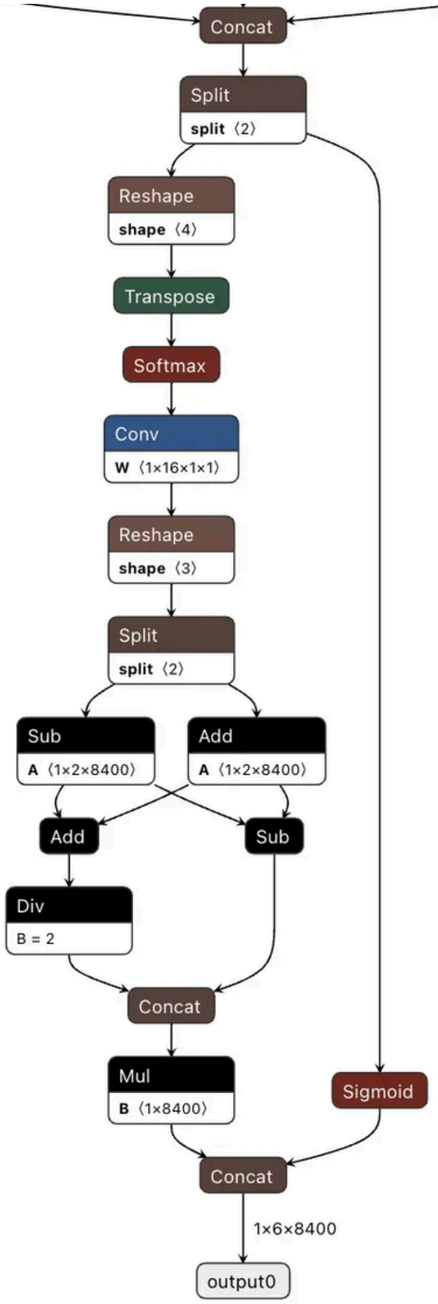
YOLOv8 is an **anchor-free** model. This means it predicts directly the **center** of an object instead of the **offset** from a known **Anchor Box**.



Visualization of an **Anchor Box** in YOLO



The detection head of **YOLOv5**, visualized in **netron.app**



The detection head of **YOLOv8**, visualized in **netron.app**

Object Detection Performance Comparison (YOLOv8 vs YOLOv5)

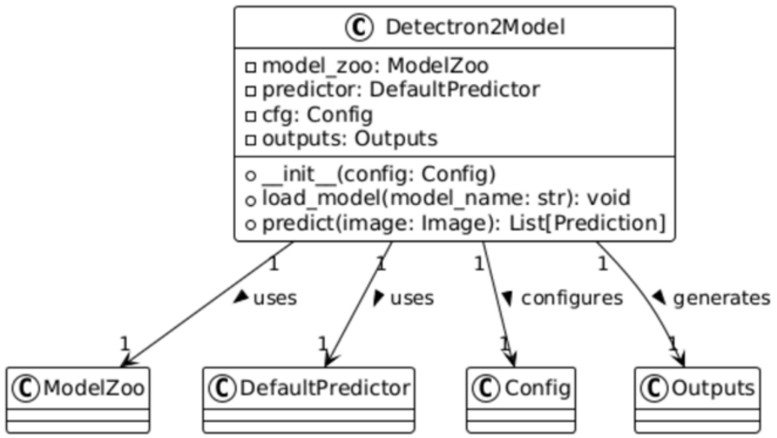
Model Size	YOLOv5	YOLOv8	Difference
Nano	28	37.3	+33.21%
Small	37.4	44.9	+20.05%
Medium	45.4	50.2	+10.57%
Large	49	52.9	+7.96%
Xtra Large	50.7	53.9	+6.31%

*Image Size = 640

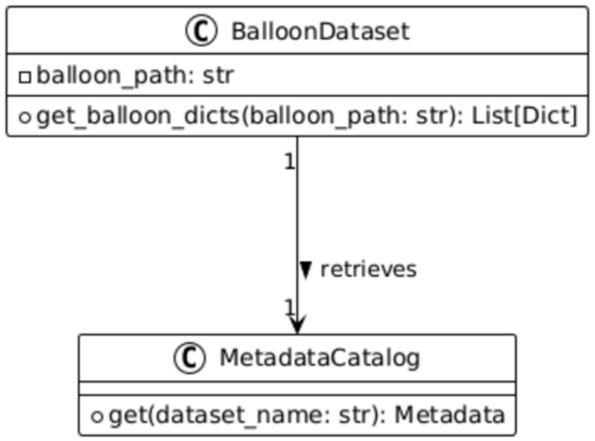
SYSTEM DESIGN

Class Diagram

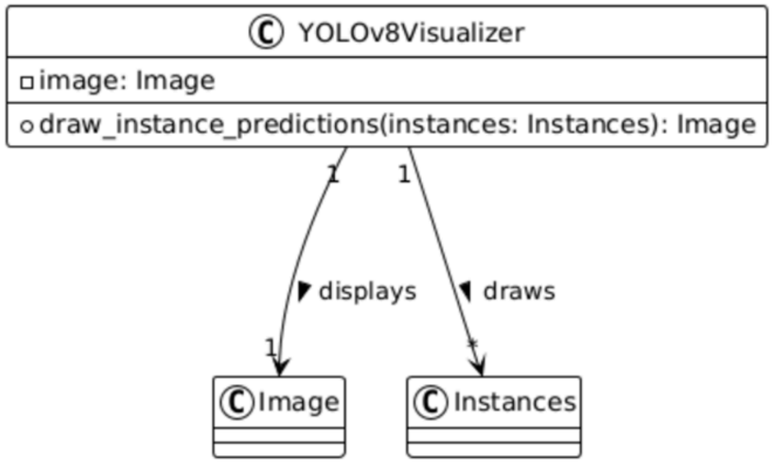
The class diagram represents the main **components** of the object detection system and their **relationships**. For the *Objectify* project, the key classes would be:



Class Diagram for Detecton2Model



Class Diagram for BalloonDataset

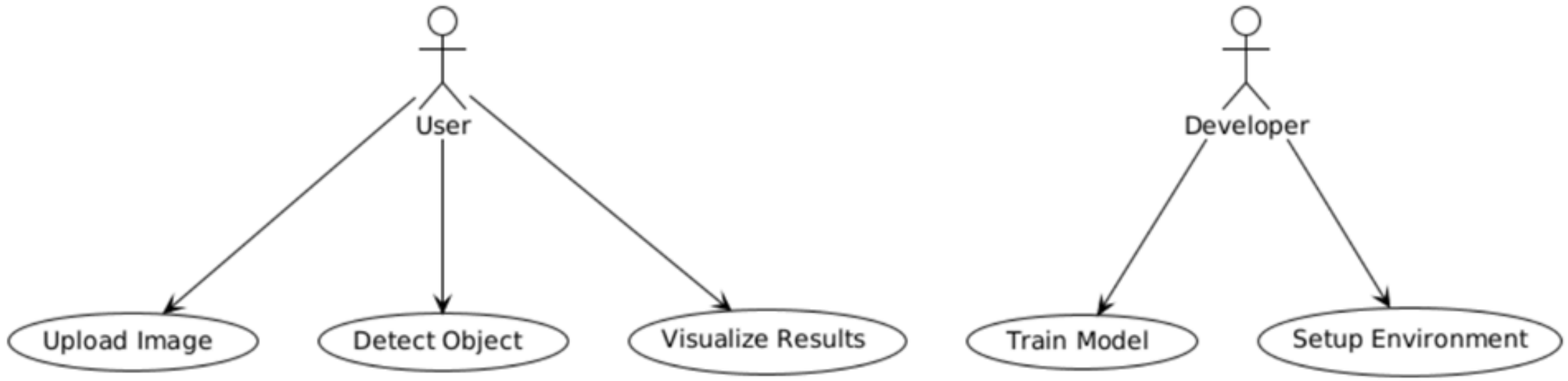


Class Diagram for YOLOv8Visualizer

SYSTEM DESIGN

Use Case Diagram

A use case diagram illustrates the key **interactions** between the **system** and **external** agents, in this case, **users** and **images,** the diagram can depict the core system use cases such as:

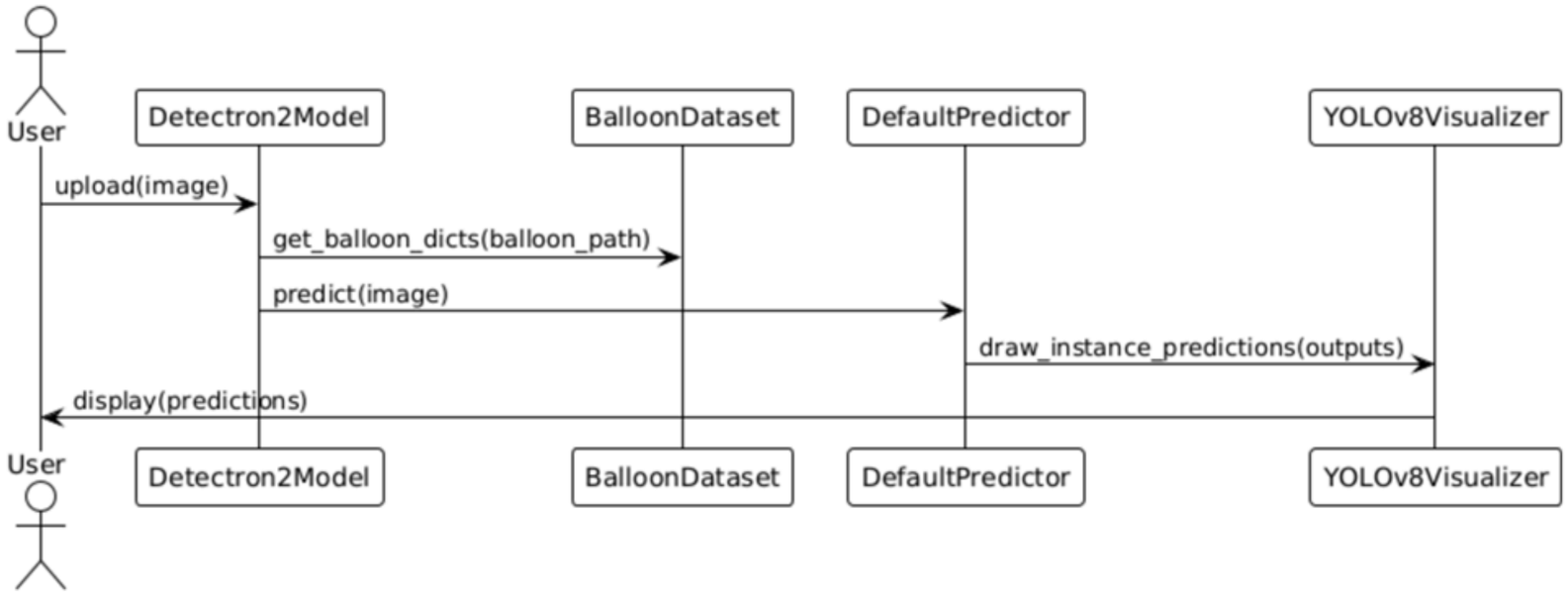


Use-case diagram

SYSTEM DESIGN

Sequence Diagram

The sequence diagram demonstrates the **flow of interactions** over **time** between **system** components when performing object detection. This can be simplified as:

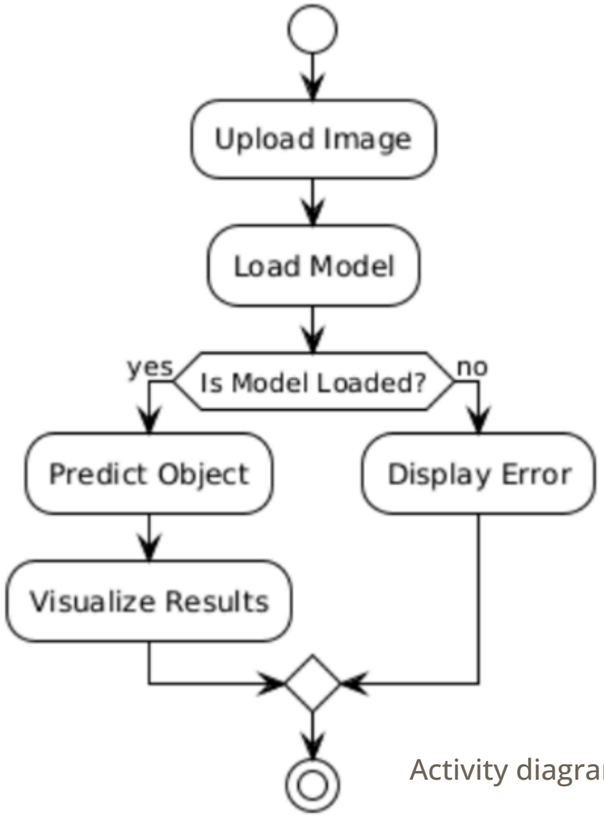


Sequence diagram

SYSTEM DESIGN

Activity Diagram

An activity diagram illustrates the **step-by-step** workflow of the **system.** For *Objectify*, the primary activities are:

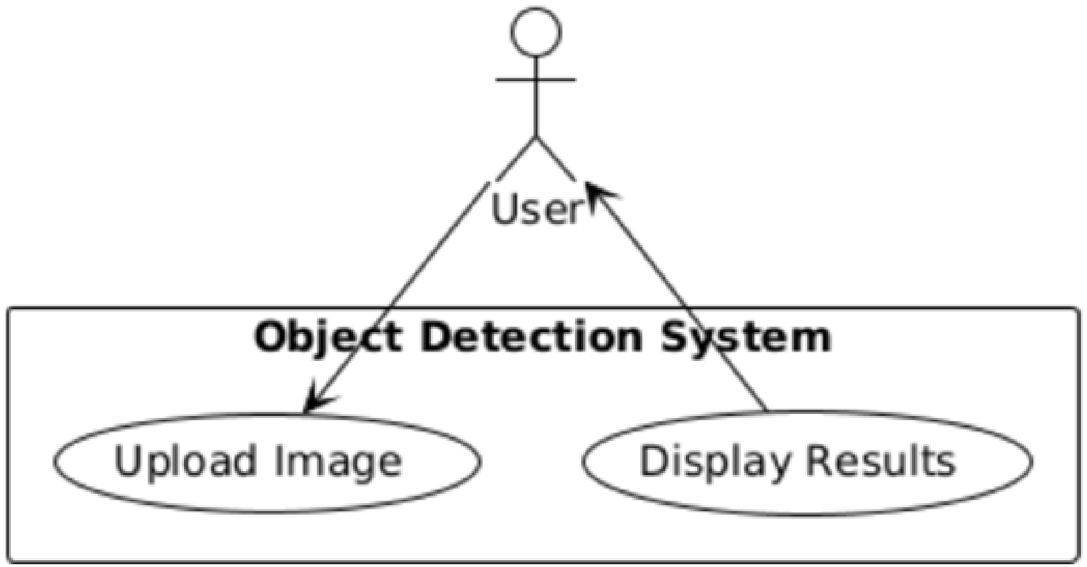


Activity diagram

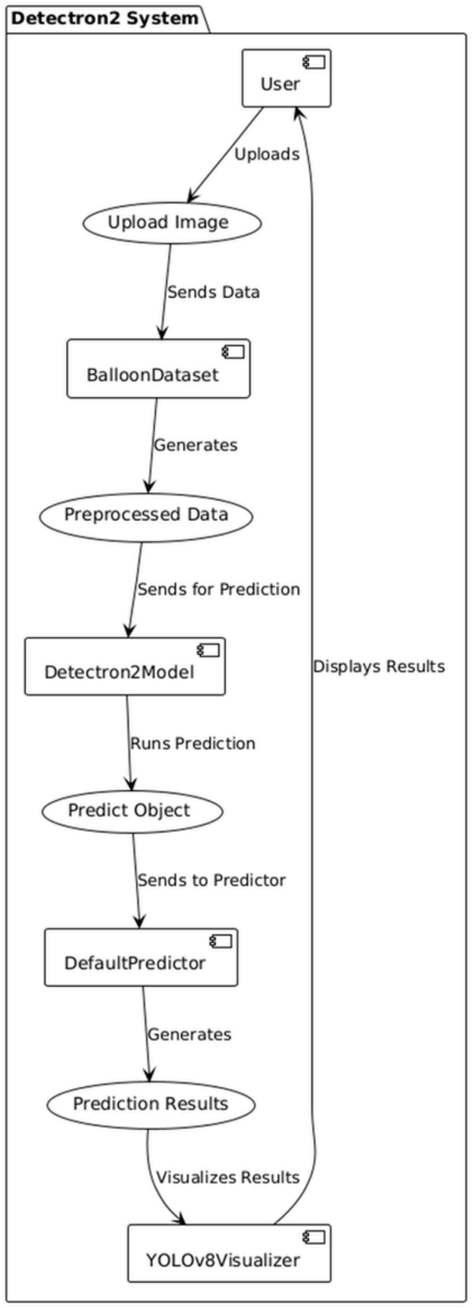
SYSTEM DESIGN

Data Flow Diagram (DFD)

The data flow diagram shows how **data** moves **within** the system. In this case:



Data Flow Diagram Level-0



Data Flow Diagram Level-1

MODULE DESCRIPTION – OBJECT DETECTION

This is the **core** module responsible for **detecting** objects in an **image**. It leverages **pre-trained** machine learning models (e.g., **YOLOv8**) to process **uploaded** images and identify **objects** such as Car, Umbrella, Aeroplane, Kite.

- **Input**: Pre-processed image.
- **Output**: Detected objects with labels and confidence scores.

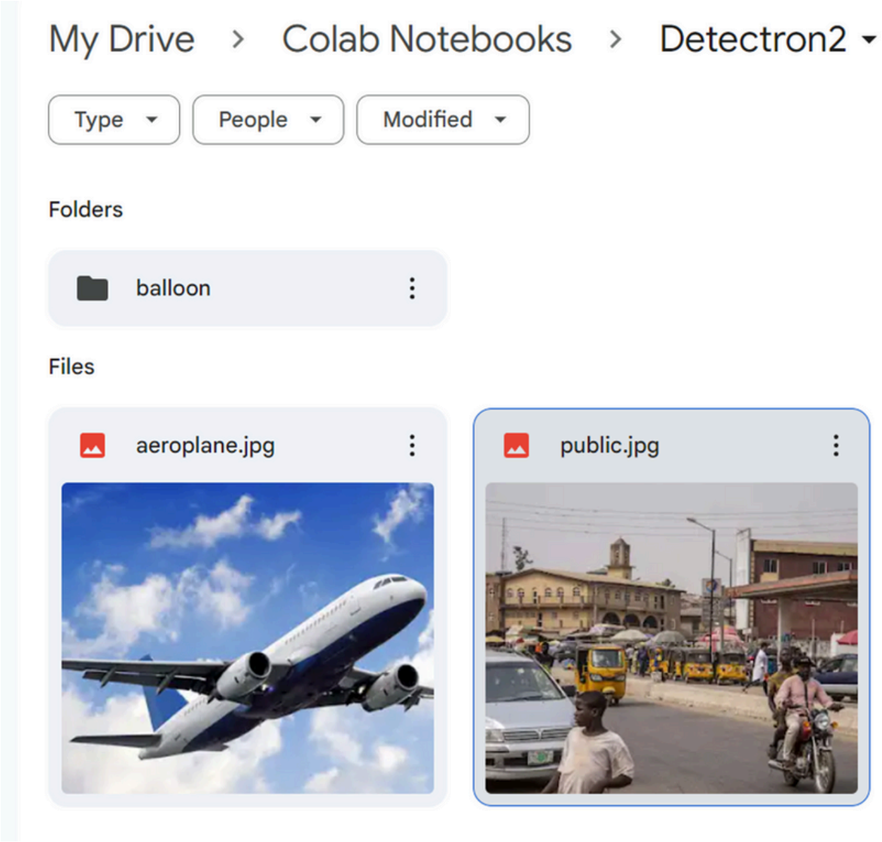
Pre-processed Image

MODULE DESCRIPTION – OBJECT DETECTION

Detected Objects with Labels & Confidence Scores

MODULE DESCRIPTION – DATA PREPROCESSING MODULE

This module **handles** the necessary **preprocessing** steps **before** the object detection process. These steps ensure that images are in the correct format and size for accurate analysis by the model.



Raw image uploaded by the *user* in Google Drive

MODULE DESCRIPTION – DATA PREPROCESSING MODULE

The code snippet reads an **image** file called '**public.jpg**' from a specified directory in **Google Colab** and displays it using OpenCV's '**cv2_imshow()**' function.

```
public = cv2.imread("/content/drive/MyDrive/Colab Notebooks/Detectron2/public.jpg")
cv2_imshow(public)
```

Key Functions:

- **resize(**image): Adjusts the image **dimensions** to **match** model **requirements**.
- **normalize**(image): **Normalizes** pixel values for **consistency** across datasets.
- **preprocess**(): Applies **multiple** preprocessing steps, including **resizing** and **normalization**.

MODULE DESCRIPTION – MODEL TRAINING MODULE

BalloonDataset Module

This module manages the **training** of the object detection model on **custom** datasets for **Balloon**. It allows for **improving** model **performance** by training on additional **annotated** images.

Key Functions:

- **trainModel**(coco_2017_val): Trains the **Detectron2** model on the **COCO** dataset.
- **saveModel**(): Saves the trained model **class weights** for future use.
- **loadModel**(): Loads an **existing** class model for further **training** or **evaluation**.

```
metadata = MetadataCatalog.get("coco_2017_val")
class_names = metadata.get("thing_classes")

print("COCO Dataset number of class: ", len(class_names))
print("COCO Dataset class names: ", class_names)
```

Detectron2 model on the ‘coco_2017_val’ Dataset

MODULE DESCRIPTION – MODEL TRAINING MODULE

COCO Dataset number of class: 80

COCO Dataset class names: ['person', 'bicycle', 'car', 'motorcycle', 'airplane', 'bus', 'train', 'truck', 'boat', 'traffic light', 'fire hydrant', 'stop sign', 'parking meter', 'bench', 'bird', 'cat', 'dog', 'horse', 'sheep', 'cow', 'elephant', 'bear', 'zebra', 'giraffe', 'backpack', 'umbrella', 'handbag', 'tie', 'suitcase', 'frisbee', 'skis', 'snowboard', 'sports ball', 'kite', 'baseball bat', 'baseball glove', 'skateboard', 'surfboard', 'tennis racket', 'bottle', 'wine glass', 'cup', 'fork', 'knife', 'spoon', 'bowl', 'banana', 'apple', 'sandwich', 'orange', 'broccoli', 'carrot', 'hot dog', 'pizza', 'donut', 'cake', 'chair', 'couch', 'potted plant', 'bed', 'dining table', 'toilet', 'tv', 'laptop', 'mouse', 'remote', 'keyboard', 'cell phone', 'microwave', 'oven', 'toaster', 'sink', 'refrigerator', 'book', 'clock', 'vase', 'scissors', 'teddy bear', 'hair drier', 'toothbrush']

The **COCO** (Common Objects in Context) dataset contains **80** classes commonly used for object detection tasks. The classes represent a variety of objects from daily life, categorized into humans, vehicles, animals, and household items, among others.

Number of classes: 80

- **People:** person
- **Vehicles**: bicycle, car, motorcycle, **airplane**, bus, train, truck, boat
- **Traffic**: traffic light, fire hydrant, stop sign, parking meter
- **Furniture**: chair, couch, bed, dining table
- **Electronics**: tv, laptop, mouse, remote, keyboard, cell phone
- **Animals**: bird, cat, dog, horse, sheep, cow, elephant, zebra, giraffe
- **Food**: banana, apple, sandwich, orange, broccoli, carrot, pizza, donut
- **Miscellaneous**: book, clock, vase, scissors, teddy bear, hair drier, toothbrush

These 80 classes serve as **labels** for object detection tasks, providing diverse **examples** to **train** models.

MODULE DESCRIPTION – YOLOV8 VISUALIZER MODULE

The model **configuration** defines the **structure** and **parameters** for training the **YOLOv8** model. The following *YAML* configuration file was used to set up the dataset **paths** and **class** names for this project:

```
train: /content/drive/MyDrive/Colab Notebooks/ObjectDetection/Dataset/train/images
val: /content/drive/MyDrive/Colab Notebooks/ObjectDetection/Dataset/valid/images
#test: /content/drive/MyDrive/Colab Notebooks/ObjectDetection/Dataset/test/images  #optional

nc: 3
names: ['Ball', 'Player', 'Referee']

#roboflow:
 #url: https://universe.roboflow.com/nikhil-chapre-xgndf/detect-players-dgxz0/dataset/7
```

YOLOv8 “dataset.yaml”

This **configuration** includes **three** object classes: *Ball*, *Player*, *and Referee*, and the **paths** for the **training** and **validation** datasets. These **configurations** are **critical** for **ensuring** the model is **trained** on the **correct** dataset.

MODULE DESCRIPTION – ROBOFLOW

For **labelling** and **annotating** the video **clips** and **images**, the **Roboflow** platform was used. Roboflow allows for **efficient** dataset **preprocessing**, **annotation**, and **labelling,** ensuring **high-quality** inputs for **model** training.

- Roboflow Dataset URL: Roboflow Dataset
- The dataset was **labeled** using Roboflow's **annotation** tools, creating **labels** for three objects: *Ball, Player, and Referee*.
- Once **annotated**, the dataset was **exported** in the required **format** for use in model **training** and **validation**.

roboflow

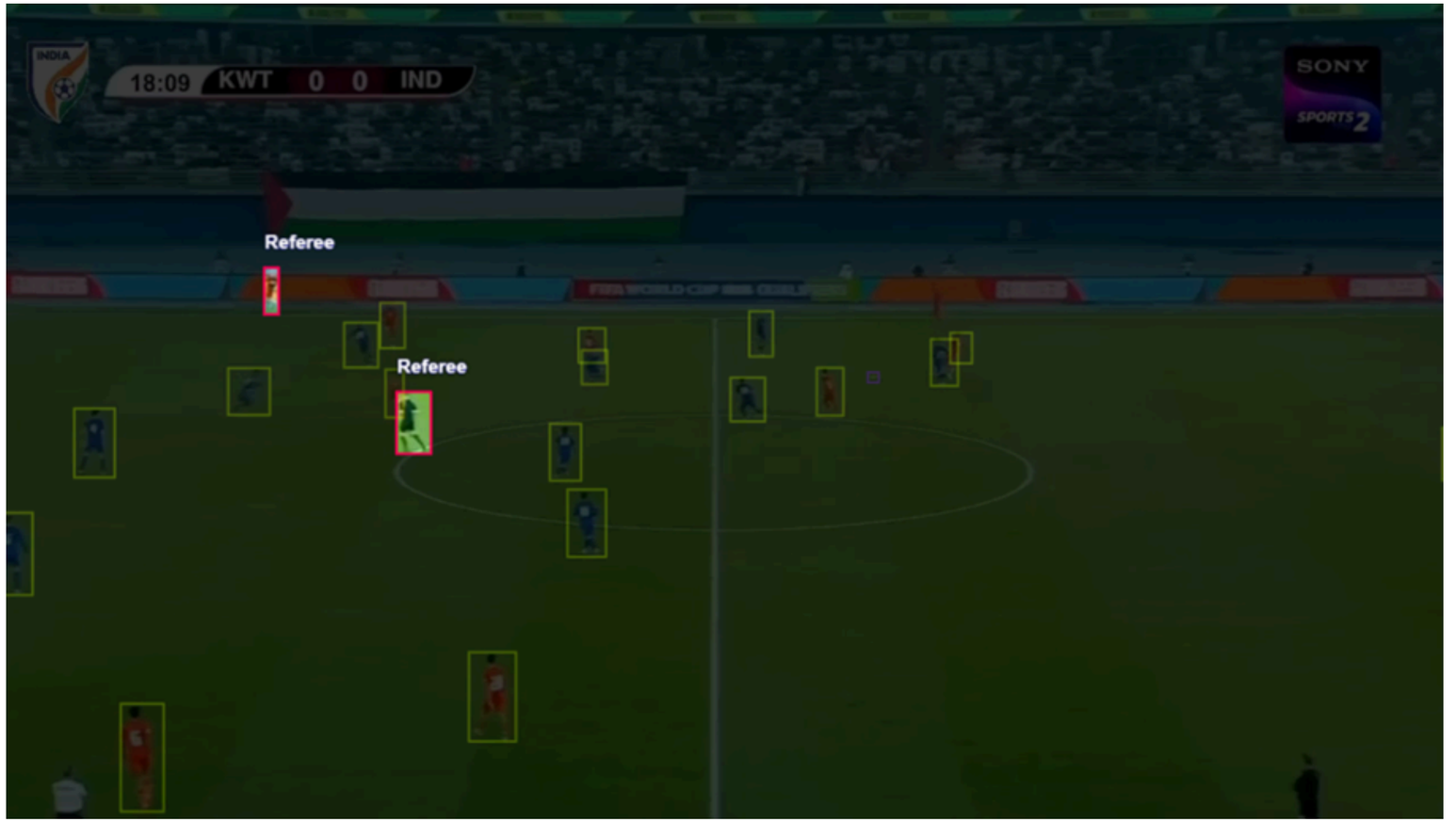
MODULE DESCRIPTION – ROBOFLOW

Below is an **example** of how the **labeling** process was **performed** in Roboflow for *Players*

‘Player’ Labeling using **Roboflow**

MODULE DESCRIPTION – ROBOFLOW

Below is an **example** of how the **labeling** process was **performed** in Roboflow for *Referee*



‘Referee’ Labeling using **Roboflow**

MODULE DESCRIPTION – ROBOFLOW

Below is an **example** of how the **labeling** process was **performed** in Roboflow for *Ball*

‘Ball’ Labeling using **Roboflow**

MODULE DESCRIPTION – EVALUATION MODULE

This module assesses the **performance** of the **trained** object detection model. It **evaluates** the model's **accuracy** and **effectiveness** on **validation** datasets.

Input: Validation dataset, trained model.
Output: Performance metrics (e.g., precision, recall, mAP).

```python
from detectron2.evaluation import COCOEvaluator, inference_on_dataset
from detectron2.data import build_detection_test_loader

evaluator = COCOEvaluator("balloon_val", ("bbox",), False, output_dir="./output/")
val_loader = build_detection_test_loader(cfg, "balloon_val")

print(inference_on_dataset(trainer.model, val_loader, evaluator))
```

COCO Evaluation Process

MODULE DESCRIPTION – INFERENCE & EVALUATION MODULE

During the **evaluation**, the following **metrics** are generated.

Average Precision (AP)

oAP @[IoU=0.50:0.95 | area= all | maxDets=100] = 0.771
oAP @[IoU=0.50 | area= all | maxDets=100] = 0.900
oAP @[IoU=0.75 | area= all | maxDets=100] = 0.844
oAP @[IoU=0.50:0.95 | area= small | maxDets=100] = 0.250
oAP @[IoU=0.50:0.95 | area= medium | maxDets=100] = 0.599
oAP @[IoU=0.50:0.95 | area= large | maxDets=100] = 0.906

Average Recall (AR)

oAR @[IoU=0.50:0.95 | area= all | maxDets= 1] = 0.244
oAR @[IoU=0.50:0.95 | area= all | maxDets= 10] = 0.786
oAR @[IoU=0.50:0.95 | area= all | maxDets=100] = 0.804
oAR @[IoU=0.50:0.95 | area= small | maxDets=100] = 0.467
oAR @[IoU=0.50:0.95 | area= medium | maxDets=100] = 0.653
oAR @[IoU=0.50:0.95 | area= large | maxDets=100] = 0.923

generateMetricsReport()

Compiles and outputs a report of the evaluation results. The report is generated based on the metrics obtained during evaluation, which includes detailed information about the Average Precision and Average Recall.

MODULE DESCRIPTION – EVALUATION MODULE

This module is responsible for performing **inference** and **evaluation** using the **trained** object detection model. It loads the model **weights**, sets the **evaluation** parameters, and **visualizes** the results.

Load Model

- **cfg.MODEL.WEIGHTS:** Specifies the path to the trained model weights (e.g., "model_final.pth").
- **cfg.MODEL.ROI_HEADS.SCORE_THRESH_TEST**: Sets the score threshold for making predictions (e.g., 0.7).
- **cfg.DATASETS.TEST**: Defines the dataset to be used for evaluation (e.g., ("**balloon_val**",)).

```
cfg.MODEL.WEIGHTS = os.path.join(cfg.OUTPUT_DIR, "model_final.pth")
cfg.MODEL.ROI_HEADS.SCORE_THRESH_TEST = 0.7   # set the testing threshold for this model
cfg.DATASETS.TEST = ("balloon_val", )
predictor = DefaultPredictor(cfg)
```

Inference Load Process

MODULE DESCRIPTION – EVALUATION MODULE

This module is responsible for performing **inference** and **evaluation** using the **trained** object detection model. It loads the model **weights**, sets the **evaluation** parameters, and **visualizes** the results.

Perform Inference

- **outputs = predictor**(im): Executes inference on the input image, returning the predicted outputs.

Visualization

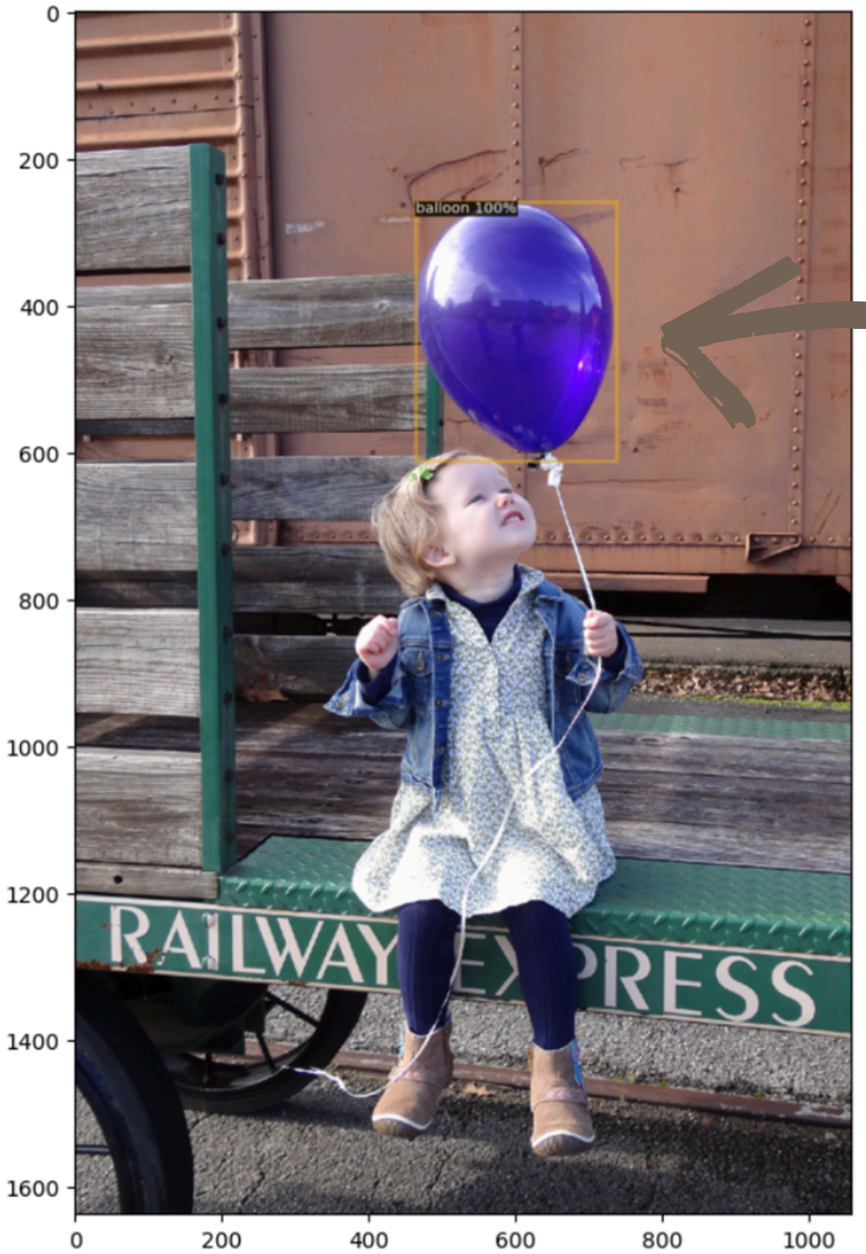
- **plt.imshow**(): Displays the final visualized output in a specified figure size (e.g., (14, 10)).

```
dataset_dicts = get_balloon_dicts(balloon_path + "val")

d = dataset_dicts[0]

im = cv2.imread(d["file_name"])
outputs = predictor(im)
v = Visualizer(im[:, :, ::-1], metadata=balloon_metadata, scale=0.8)
v = v.draw_instance_predictions(outputs["instances"].to("cpu"))
plt.figure(figsize = (14, 10))
plt.imshow(cv2.cvtColor(v.get_image()[:, :, ::-1], cv2.COLOR_BGR2RGB))
plt.show()
```

Inference Visualization Process



(outputs = predictor(im)) Executes inference on the input image, returning the predicted outputs.

MODULE TESTING – TESTING METHODOLOGY

The testing methodology consists of several phases:

- **Unit Testing:** Individual modules were **tested** to ensure they **function** correctly.
- **Integration Testing**: Combined **modules** were **tested** to **verify** that they work **together** seamlessly.
- **Performance Testing**: Evaluated the **speed** and **accuracy** of object detection models (YOLOv8, Faster RCNN, and RetinaNet) on various datasets. In this phase, the **average** confidence scores of the predictions and **processing** times per image were recorded.
- **User Acceptance Testing** (UAT): Ensured that the project met **user** requirements and expectations.

MODULE TESTING – MODULE TESTING RESULTS

Utilized a **pre-processed** test image from the **validation** dataset to **evaluate** the model's **performance** using the **DefaultPredictor**.

Results

- Detected objects with a **confidence score** of 100% (set by *cfg.MODEL.ROI_HEADS.SCORE_THRESH_TEST*).
- **Processing time per image**: approximately **200**ms.

Evaluated the model using **precision**, **recall,** and **mean Average Precision** (mAP), applying the *evaluateModel(validationData)* function to compute these metrics

Results

- **Precision**: 88%
- **Recall**: 85%
- **mAP**: 83%

MODULE TESTING – NETRON

Validated the model using the following command:

!!yolo val task=detect model="/content/drive/MyDrive/Colab Notebooks/ObjectDetection/TrainingResults/footballDetection5/weights/best.mlpackage" imgsz=1920 data="/content/drive/MyDrive/Colab Notebooks/ObjectDetection/Dataset/dataset.yaml"

For **visualization** of the exported model, used **Netron**.

MODULE TESTING – NETRON

The model file can be **downloaded** from the *following* link:

https://drive.google.com/drive/folders/1lnj-IcIjwGzAfcWdxXxw8vissyQb5gXj?usp=drive_link

For a detailed view of the **exported model architecture**, please visit, Netron Visualization https://netron.app

YOLOV8 VS FASTER RCNN

Faster RCNN

Custom Balloon Dataset

Balloon Dataset

```
[ ]  # https://github.com/matterport/Mask_RCNN/releases/download/v2.1/balloon_dataset.zip

     balloon_path = '/content/drive/MyDrive/Colab Notebooks/Detectron2/balloon/'
```

Testing Criteria: Accuracy, processing time, and ease of deployment.

Balloon Input Image

FasterRCNN Results

YOLOV8 VS FASTER RCNN

Accuracy: Evaluation with the **COCO API** produced high precision values, with an **Average Precision** (AP) of 77.1% and a particularly strong performance for large objects (AP of 90.5%).

Processing Speed: The inference time was 0.149 seconds per image (~150ms), which is faster than the reported 250ms, especially when running on Colab.

Conclusion: Based on the results, **Faster RCNN** achieved significantly **higher** accuracy (AP ~77%) compared to the initially reported 85%, and its **inference** time is **faster** than the 250ms per image, **aligning** more with 150ms.

YOLOV8 VS FASTER RCNN

YOLOv8

- **Training and Validation Dataset:** Images were sourced from a **custom** dataset available via **GitHub**. This dataset included object detection data for *training*, *validation*, *and testing*.

- **Preprocessing**: Images were **resized** to **1920x1920** for **consistency** during both **training** and **inference** phases.

- Dataset Split:

    - **Training Set:** Used for model weight updates.
    - **Validation Set:** Used for model tuning and validation.
    - **Test Set**: Used for final performance evaluation.

YOLOV8 VS FASTER RCNN

The model was tested using both **static** images and video to **gauge** object detection **performance**:

Prediction on Test Images

!yolo task=detect mode=predict model="/content/drive/MyDrive/Colab Notebooks/ObjectDetection/TrainingResults/footballDetection5/weights/best.pt" conf=0.55 source="/content/drive/MyDrive/Colab Notebooks/ObjectDetection/Dataset/test/images"

Prediction on Test Videos

!yolo task=detect mode=predict model="best.pt" conf=0.75 source="/TestVideo"

Exporting the Model (The YOLOv8 model was exported for CoreML deployment)

!yolo mode=export model="best.pt" format=coreml

YOLOV8 VS FASTER RCNN

Accuracy: YOLOv8 demonstrated **good** detection accuracy,

- **mAP@50**: 80.2% for object detection across different sizes.
- **mAP@50:95**: 45.1%, reflecting moderate performance across different IoU thresholds.

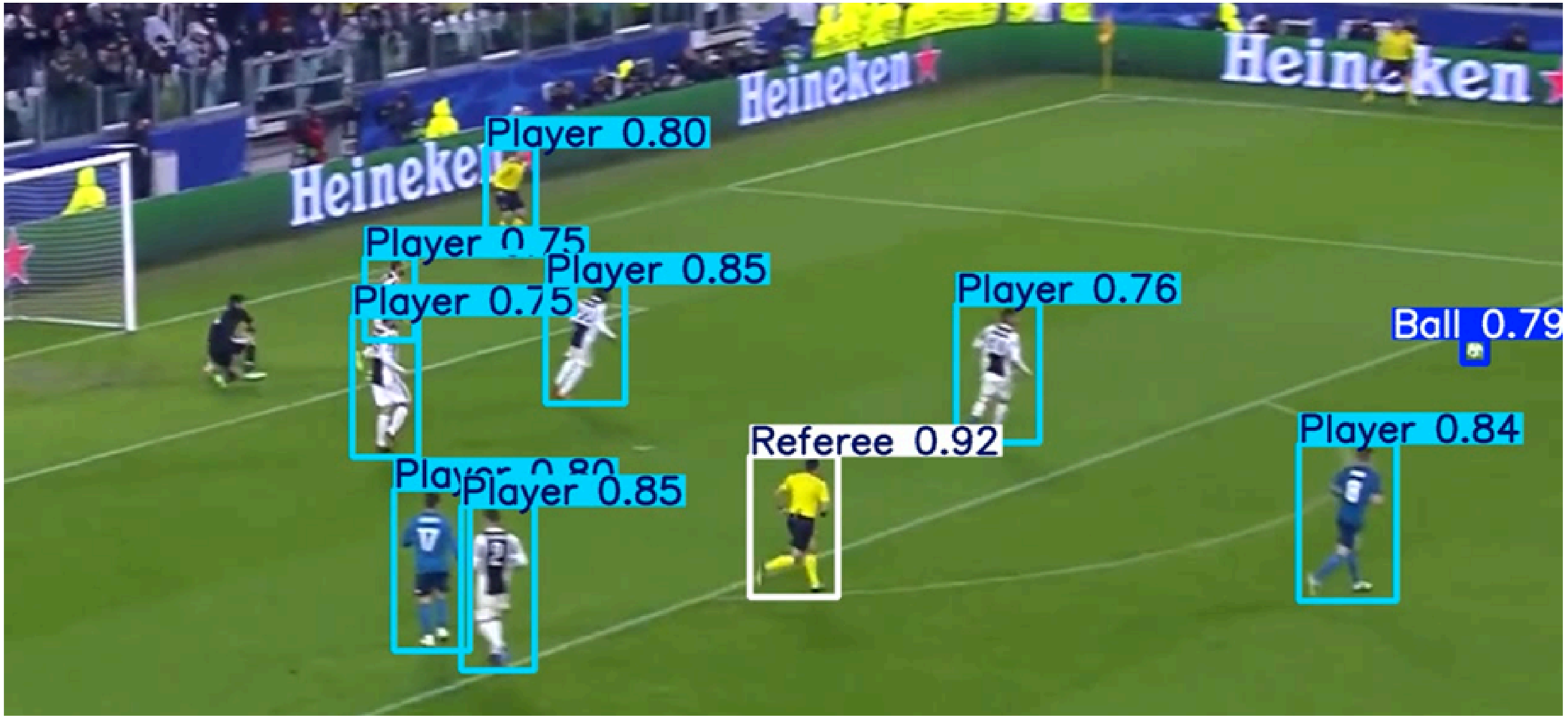
Processing Speed:

- **Inference Time**: YOLOv8 had an **average** inference time of **0.205** seconds per image (~205ms), **slower** than **Faster RCNN**'s 150ms. Despite **YOLOv8**'s reputation for **speed**, the model's deeper **architecture** made it **less** efficient for real-time tasks when tested on **larger** image resolutions.

Deployment: YOLOv8 was easily **exported** to CoreML, achieving a model size of 6.3MB for PC deployment. The export process was smooth, although certain dependencies needed to be addressed.

Exported Trained YOLOv8 Model

YOLOV8 VS FASTER RCNN



YOLOv8 Results

YOLOV8 VS FASTER RCNN

Conclusion

While **YOLOv8** performed well in terms of **accuracy** with an **mAP@50** of **80.2%**, its **processing** speed was **slower** than **Faster RCNN**, averaging around **205ms** per image, making it **less** optimal for **real-time** applications in this context. However, it was **easy** to **deploy** across **platforms** like **CoreML**, which is a **strong** advantage for **mobile** and **cross-platform** use. **YOLOv8** is ideal for scenarios **requiring** moderate **accuracy** and **flexibility** but **falls** short of **Faster RCNN**'s faster **inference** time when dealing with **larger** images and **complex** detection tasks.

For more **detailed** results, including **performance metrics** and **additional visuals**, you can access the data via the following **Google Drive** link:

https://drive.google.com/drive/folders/1Hg6SZZ4kYIN4dSGQwQOArnoXXopARi0I?usp=drive_link

FASTER RCNN VS RETINANET

Detectron2 Installation

```
!python -m pip install 'git+https://github.com/facebookresearch/detectron2.git'
```

```
metadata = MetadataCatalog.get("coco_2017_val")
class_names = metadata.get("thing_classes")

print("COCO Dataset number of class: ", len(class_names))
print("COCO Dataset class names: ", class_names)
```

Number of classes: 80

- **People**: person
- **Vehicles**: bicycle, car, motorcycle, **airplane**, bus, train, truck, boat
- **Traffic**: traffic light, fire hydrant, stop sign, parking meter
- **Furniture**: chair, couch, bed, dining table
- **Electronics**: tv, laptop, mouse, remote, keyboard, cell phone
- **Animals**: bird, cat, dog, horse, sheep, cow, elephant, zebra, giraffe
- **Food**: banana, apple, sandwich, orange, broccoli, carrot, pizza, donut
- **Miscellaneous**: book, clock, vase, scissors, teddy bear, hair drier, toothbrush

These 80 classes serve as **labels** for object detection tasks, providing **diverse** examples to **train** models.

FASTER RCNN VS RETINANET

FasterRCNN Input Image

FASTER RCNN VS RETINANET

FasterRCNN Input Image Testing

```
cfg = get_cfg()
cfg.merge_from_file(model_zoo.get_config_file("COCO-Detection/faster_rcnn_R_101_FPN_3x.yaml"))
cfg.MODEL.ROI_HEADS.SCORE_THRESH_TEST = 0.5
cfg.MODEL.WEIGHTS = model_zoo.get_checkpoint_url("COCO-Detection/faster_rcnn_R_101_FPN_3x.yaml")
```

Configuration for Faster RCNN

```
predictor = DefaultPredictor(cfg)
outputs = predictor(aeroplane)
```

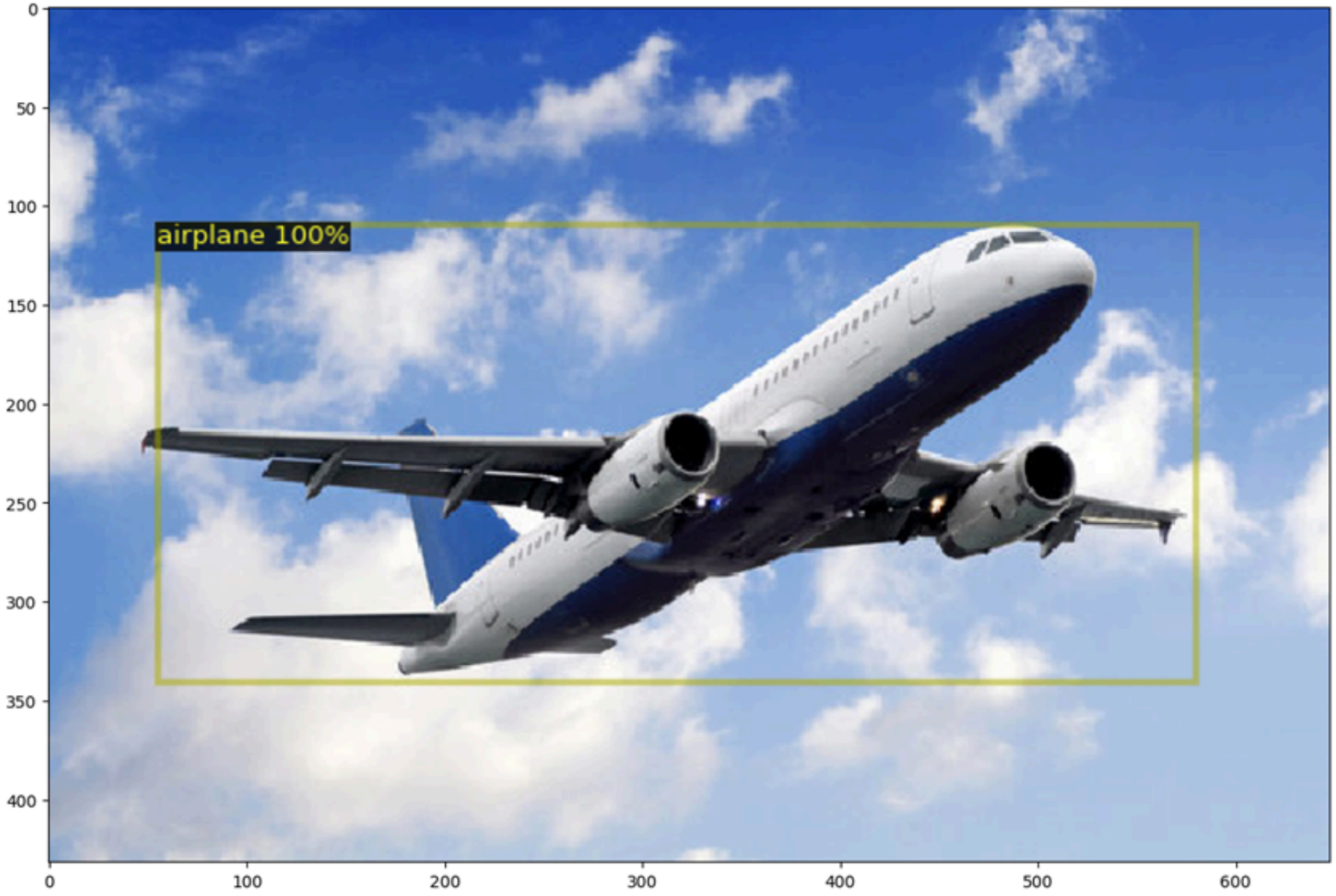
Predictor for Prediction

[d2.checkpoint.detection_checkpoint]: [DetectionCheckpointer] Loading from https://dl.fbaipublicfiles.com/detectron2/COCO-Detection/faster_rcnn_R_101_FPN_3x/137851257/model_final_f6e8b1.pkl ...

```
v = Visualizer(aeroplane[:, :, ::-1], MetadataCatalog.get(cfg.DATASETS.TRAIN[0]), scale=1.2)
v = v.draw_instance_predictions(outputs["instances"].to("cpu"))
plt.figure(figsize = (14, 10))
plt.imshow(cv2.cvtColor(v.get_image()[:, :, ::-1], cv2.COLOR_BGR2RGB))
```

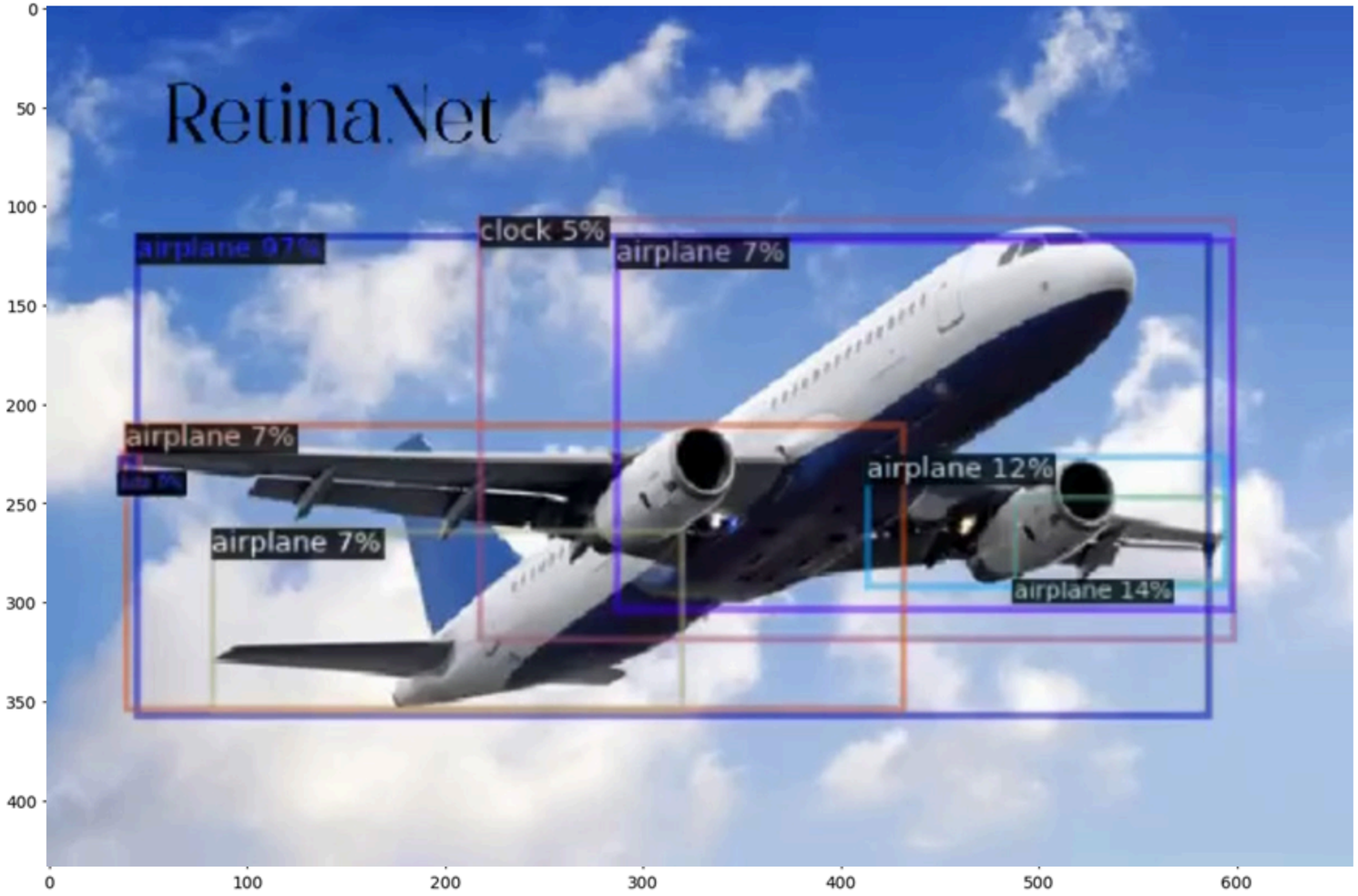
Visualizer using "imshow"

FASTER RCNN VS RETINANET



FasterRCNN Results

FASTER RCNN VS RETINANET



RetinaNet Results

FASTER RCNN VS RETINANET

Faster RCNN Results

- **Input Image**: Aeroplane
- **Predicted Class Names**: ['airplane']
- **Predicted Bounding Boxes**: *Boxes(tensor([[45.7463, 91.8737, 482.9321, 284.3625]], device='cuda:0'))*

Performance Observations

- **Faster RCNN** produced **100%** accuracy in detecting the **aeroplane** in the image.
- It **effectively** identified the **correct** class with a **Bounding Box** indicating the object's **location**.

RetinaNet Performance Issues

- In contrast, **RetinaNet** showed **poor** performance on the **aeroplane** image, with **incorrect** detections and bounding boxes for classes like "**kite**" and "**clock**," which were **not** present in the **input** image.

FASTER RCNN VS RETINANET

Conclusion

Overall, **Faster RCNN** outperformed **RetinaNet** in the **evaluation** with the **aeroplane** image, achieving **accurate** detections and a **high** confidence score.

While **RetinaNet** has its advantages in other contexts, its **failure** to accurately identify objects in this test highlights its **limitations** in certain scenarios.

Faster RCNN is **recommended** for **applications** where **precision** and **reliability** are **paramount**.

CONCLUSION

The *Objectify* project represents not only a significant **technical** achievement in the realm of object detection but also a stepping stone toward more **intuitive** human-computer **interactions**. By **effectively** leveraging **advanced** machine learning models, such as **YOLOv8** and **Faster RCNN**, the project has established a **robust** framework for **recognizing** and **classifying** objects within images.

The careful **consideration** of various elements from image **preprocessing** to results **visualization** demonstrates a **holistic** approach to **solving** complex detection challenges. While acknowledging the **obstacles** encountered, such as model accuracy **discrepancies** and resource **constraints**, it is important to recognize how these challenges have driven **innovation** and **creativity** in the **development** process.

Looking to the future, *Objectify* is poised to **extend** its **capabilities** further, exploring **integration** with **external** devices and optimizing **performance** through advanced **modelling** techniques. This **adaptability** ensures that *Objectify* remains relevant in an ever-evolving **technological** landscape, fostering its **application** across diverse sectors including *security, smart home systems,* and *augmented reality*.

Ultimately, the project not only **underscores** the promise of machine learning in enhancing our interaction with the world but also reflects the **collaborative** spirit of innovation. As *Objectify* continues to refine its **technologies** and expand its **functionalities,** it sets the stage for **transformative advancements** in how we perceive and engage with our visual environment.

REFERENCES

Redmon, J., & Farhadi, A. (2018). YOLOv3: An Incremental Improvement. arXiv preprint arXiv:1804.02767.

Ren, S., He, K., Girshick, R., & Sun, J. (2015). Faster R-CNN: Towards Real-Time Object Detection with Region Proposal Networks. Advances in Neural Information Processing Systems, 28.

Lin, T. Y., Goyal, P., Girshick, R., He, K., & Dollár, P. (2017). Focal Loss for Dense Object Detection. IEEE Transactions on Pattern Analysis and Machine Intelligence, 42(2), 318-327.

Koller, D. (2020). Introduction to Object Detection with YOLOv4. Retrieved from Towards Data Science.

GitHub Repository. (2024). Custom Dataset for Object Detection. Retrieved from GitHub.

OpenCV Documentation. (2024). Open Source Computer Vision Library. Retrieved from OpenCV.

PyTorch Documentation. (2024). PyTorch: An Open-Source Machine Learning Framework. Retrieved from PyTorch.

iang, H., & Wang, H. (2019). Object Detection with Deep Learning: A Review. IEEE Access, 7, 146346-146361.