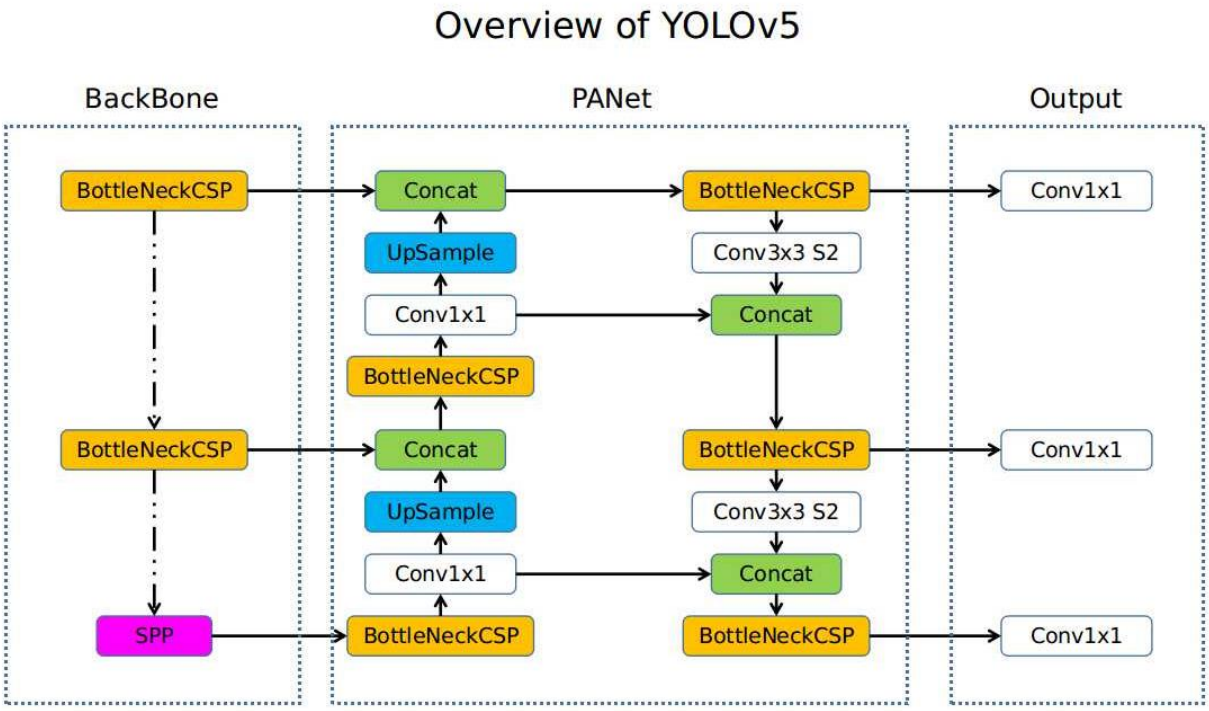
1. Overview of YOLOv5

1.1. Architecture



1. Input Layer

- Prepares the input image by resizing and normalizing it to the appropriate dimensions (e.g., 640x640).
- The input is then converted into a tensor for processing.

2. Backbone

- Extracts features from the input image.
- Common backbone: **CSPDarknet53** (Cross-Stage Partial Network), which enhances feature propagation and reduces computation.
- Key layers in the backbone:
    - **Conv2D**: Standard convolutional layers.

- **BatchNorm2D**: Normalization for faster convergence.
- **Leaky ReLU/SiLU**: Activation functions.
- **Residual Bottlenecks**: Improve gradient flow and efficiency.
- **SPPF (Spatial Pyramid Pooling Fast)**: Aggregates spatial features at different scales.

3. Neck

- Bridges the backbone and the head, focusing on fusing features from different scales for better detection.
- Key components:
  - **PANet (Path Aggregation Network)**: Enhances information flow between layers.
  - **Convolutions and Upsampling**: Integrates features from high and low resolutions.
  - **Concatenation and Fusion Layers**: Combine feature maps from different scales.

4. Head

- Outputs predictions for bounding boxes, objectness scores, and class probabilities.
- Key components:
  - **Detect Layer**: Processes the feature maps and makes final predictions.
  - **Anchor Boxes**: Predefined shapes for bounding box proposals, optimized during training.
  - Sigmoid activation functions: Used to compute confidence scores and class probabilities.

5. Loss Layer (during Training)

- Computes the loss, which combines:
  - **Bounding Box Loss**: Measures accuracy of predicted bounding boxes.
  - **Objectness Loss**: Evaluates confidence in detections.
  - **Classification Loss**: Assesses the accuracy of class predictions.

6. Post-Processing (during Inference)

- Applies:
    - **Non-Maximum Suppression (NMS)**: Removes duplicate bounding boxes.
    - **Thresholding**: Filters out low-confidence predictions.

1.2. What is YOLO?

- **You Only Look Once (YOLO)** is a state-of-the-art, real-time object detection algorithm
- Some of the reasons why YOLO is leading the competition include its:

Speed	- YOLO is extremely **fast** - It can process images at 45 Frames Per Second (FPS). - YOLO reaches more than twice the mean Average Precision (mAP) compared to other real-time systems, ⇨ makes it a great candidate for real-time processing.
High detection accuracy	High accuracy, with very few background errors.
Good generalization	providing better generalization for new domains, which makes it great for applications relying on fast and robust object detection.
Open-source	

[Source: https://www.datacamp.com/blog/yolo-object-detection-explained]

1.3. Difference between YOLOv5 and ResNET

Feature	**YOLOv5**	**ResNet**
Purpose	Object detection (localizes and classifies objects in images).	Image classification (predicts one label for an image).
Architecture	Combines feature extraction with bounding box regression and classification.	Primarily a feature extractor with residual connections for classification tasks.
Speed	Optimized for real-time detection.	Not optimized for real-time detection.

Output	Bounding boxes, confidence scores, and classes.	Class probabilities.
Use Case	Multi-object detection in images/videos.	Single-object classification in images.

2. Train custom dataset with YOLOv5.

See YOLOv5 Docs for additional details.

2.1. Create manual Dataset

a) Download Kaggle dataset

- The dataset composed of 1000 training images of traffic, vehicles and number plates, and CCTV footage videos
- These images are used for both training and validation

Labels:

- The images were labeled (using tool **LabelImg**) under 7 classes – Car, Number Plate, Blur Number Plate, Two Wheeler, Auto, Bus, and Truck in **YOLOv5** format
- with one *.txt file per image (**YOLO format**). The *.txt file specifications are:
  - One row per detected object
  - Each row is class x_center y_center width height format.
  - Box coordinates must be in normalized xywh format (from 0 to 1). If your boxes are in pixels, divide x_center and width by image width, and y_center and height by image height.

b) **Create dataset.yaml** (YOLOv5 YAML configuration file)

The dataset config file defines:

1) the paths to train / val / test image directories
2) a class names dictionary

-

2.2. Select a Model:

Select a pretrained model to start training from.
Here we select YOLOv5s, **the second-smallest** and **fastest** model available

2.3. Training:

- Train a YOLOv5s model by specifying dataset, **batch-size**, **image size** and weight --weights yolov5s.pt (pretrained).
- Training times for YOLOv5n/s/m/l/x are 1/2/4/6/8 days on a V100 GPU (Multi-GPU times faster). Use any --batch-size below. Batch sizes shown for V100-16GB.

```
python train.py --data coco.yaml --epochs 300 --weights '' --cfg yolov5n.yaml  --batch-size 128
                                                                 yolov5s                    64
                                                                 yolov5m                    40
                                                                 yolov5l                    24
                                                                 yolov5x                    16
```

- **Tip:** Add **--cache ram** or **--cache disk** to speed up training

- The commands below are used for our project.

```
!python train.py --img 640 --batch 16 --epochs 20 --data dataset.yaml --weights yolov5s.pt --cache
```

[Source: https://github.com/ultralytics/yolov5/wiki/Train-Custom-Data

3. Code explained

a) Setup Dataset

- Downloads the dataset "Traffic Vehicles Object Detection" from Kaggle using the KaggleHub tool.
- Creates a directory **/content/kaggle** to store the dataset.
- Copies the dataset to the Colab environment.

```
#download the data
import kagglehub

%mkdir kaggle
kagglehub.dataset_download("saumyapatel/traffic-vehicles-object-detection")
%cp "/root/.cache/kagglehub/datasets/saumyapatel/traffic-vehicles-object-detection/versions/1/Traffic Dataset" /content/kaggle/ -a
```

b) Setup YOLOv5 Environment

- Clone the YOLOv5 repo and install dependencies (listed in requirements.txt) for training and inference.

```
#we use yolov5 pretrained model for this project
!git clone https://github.com/ultralytics/yolov5
%cd /content/yolov5/

#install the dependencies
!pip install -r requirements.txt
```

How It Works:

- YOLOv5 is a prebuilt repository that provides an implementation of the YOLO object detection framework.
- Dependencies include libraries like PyTorch, torchvision, and others required for model training and inference.

c) Dataset Configuration

The YOLOv5 reads this YAML file to locate data and understand the dataset structure

```
#create the dataset with train and validate data
yaml_content = """
train: /content/kaggle/Traffic Dataset/images/train
val: /content/kaggle/Traffic Dataset/images/val

nc: 7
names: ['Car', 'Number Plate', 'Blur Number Plate', 'Two Wheeler', 'Auto', 'Bus', 'Truck']
"""

yaml_path = '/content/yolov5/dataset.yaml'

with open(yaml_path, 'w') as file:
    file.write(yaml_content)
```

- **yaml_content**: Defines the paths for training and validation datasets.
  - train and val: Paths to training and validation image directories.
  - nc: Number of classes in the dataset (7 in this case).
  - names: Names of the classes (e.g., Car, Bus, Truck).

- **Writes YAML File**: Creates a configuration file dataset.yaml, used during training to reference the dataset.

d) Training the Model

```
#train the model with our custom data
%cd /content/yolov5/
!python train.py --img 640 --batch 16 --epochs 20 --data dataset.yaml --weights yolov5s.pt --cache
```

- **train.py**: The script trains the YOLOv5 model using the specified parameters.
- **Arguments**:
  - --img 640: Resizes images to 640x640 pixels for training.
  - --batch 16: Processes 16 images in each batch.
  - --epochs 20: Trains the model for 20 iterations over the dataset.
  - --data dataset.yaml: Specifies the dataset configuration file.
  - --weights yolov5s.pt: Uses a pre-trained YOLOv5 small model. Pre-trained weights are fine-tuned on the new dataset.
  - --cache: Caches the dataset in memory to speed up training

Reason for Choice of these parameters:

The combination of these parameters is carefully selected to balance accuracy, computational efficiency, and resource constraints

- **Image Size** (**--img 640**)**:** Resizes input images to 640x640 pixels for uniformity during training.
  - **Answer 1:** 640x640 image resolution ensures objects like number plates and two-wheelers are detected accurately**.**
  - **Answer 2:** It's large enough to capture details but small enough to keep training efficient. For traffic analysis, we need that balance because we care about ***accuracy*** *and* ***speed***.

- **Batch size (--batch 16)**
  - **Answer 1:** It determines the number of images processed in a single training step (in this case it processes 16 images per training step)

  Since Batch size impacts both training speed and memory usage. A batch size of 16 is chosen because it fits well within the memory constraints of hardware environments such as Google Colab.
  - **Answer 2:** If we use a small batch size, like 4, training will take forever. But if we go too big, like 64, it might crash because it'll eat up all the memory.

    So, with 16, it's manageable for our system (like Google Colab), and it gives the model stable, consistent updates while training.

- **20 epochs (--epochs 20**).
  - **Answer 1: epoch** decides how many times the model should learn from the dataset (in this case is 20 times). Since we're starting with pretrained weights (yolov5s.pt), the model already knows a lot. We're just "fine-tuning" it to our traffic dataset.
  - 20 epochs are enough to teach the model only 7 classes—cars, buses, bikes, etc.—without overtraining it. Too many epochs might

make it memorize the dataset, and we don't want that. We need it to generalize to new images.

- **Answer 2:** Training for 20 epochs is a starting point for observing how well the model learns the task. Excessive epochs can lead to overfitting, especially when fine-tuning pretrained weights

Since YOLOv5 is pretrained on a large dataset (e.g., COCO), fine-tuning on 20 epochs is usually sufficient for smaller datasets like the traffic dataset.

- **Pretrained weights (yolov5s.pt) .**
  - This parameter loads pretrained weights from **the YOLOv5 small model** (yolov5s). This model is a lightweight and efficient version of YOLOv5, designed for faster training and inference.

  By starting with pretrained weights, the model benefits from general knowledge of object detection, requiring less training time to specialize in our dataset (fine-tuning). The choice of the yolov5s variant is a balance of resource-efficient and accuracy.

- **Cache (--cache)**
  - This option caches the images in memory during training, reducing the need to read them repeatedly from disk.
  - This accelerates the training process, especially in environments like Google Colab, where disk I/O can be a bottleneck. Caching ensures that the model has faster access to data, enabling smoother training cycles

What is fine-tuning?

It is a machine learning technique used to adapt a pre-trained model to a new, specific task or dataset. It involves taking a model that has already been trained on a large, general-purpose dataset and then retraining it on a smaller, task-specific dataset.

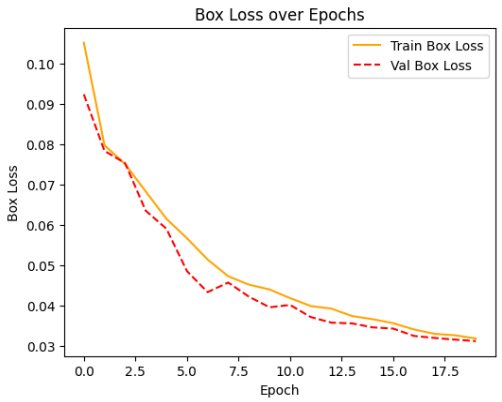
e) H

```
#save the trained weights
!cd /content/
dir_to_zip = '/content/yolov5/' #@param {type: "string"}
output_filename = 'yolov5.zip' #@param {type: "string"}
delete_dir_after_download = "No"  #@param ['Yes', 'No']

os.system( "zip -r {} {}".format( output_filename , dir_to_zip ) )

if delete_dir_after_download == "Yes":
    os.system( "rm -r {}".format( dir_to_zip ) )

files.download( output_filename )

#load the trained weights to the current directory
%cd /content
!unzip yolov5.zip -d /content
%cd /content/yolov5/

!cd /content/
!rm -r yolov5
```

f) Visualizing Results

- **Reads CSV**: Loads training metrics like box loss from **`results.csv`**.
- **Plots Graph**: Visualizes training and validation loss over epochs.

```
#Model Result Visualization
data = pd.read_csv('runs/train/exp/results.csv')
plt.figure(figsize=(20, 10))
plt.subplot(2, 3, 2)
plt.plot(data['               epoch'], data['      train/box_loss'], label='Train Box Loss', color='orange')
plt.plot(data['               epoch'], data['        val/box_loss'], label='Val Box Loss', linestyle='--', color='red')
plt.xlabel('Epoch')
plt.ylabel('Box Loss')
plt.title('Box Loss over Epochs')
plt.legend()
plt.show()
```



We observe that the **training box loss** and **validation box loss** decrease as the epochs increase, it means the model is improving over time and learning better.

- Box loss refers to the difference between the predicted bounding boxes and the actual bounding boxes. The box loss is specifically concerned with how accurately the predicted boxes match the ground truth boxes.

- When both the training and validation box losses decrease, it means that the model is becoming more accurate at predicting the bounding boxes as it learns from the data during training.
- If the loss increases at some point, it might suggest the model has reached a point of diminishing returns, or you might need to adjust training parameters, to continue improving.

g) Evaluation the model

```
#evaluating the model
!python val.py --img 640 --data dataset.yaml --weights runs/train/exp/weights/best.pt
```

```
val_result = Image(filename='runs/val/exp/confusion_matrix.png')
display(val_result)
```

h) C

- **Inference**: Loads the trained weights for testing.

```
#load the trained model
model = torch.hub.load("./", "custom", path="./runs/train/exp/weights/best.pt", source="local")
```

i) Testing on Images:

```
#Testing on 3 examle
img_arr = [244, 175, 326]
for img in img_arr:
    img = f'/content/kaggle/Traffic Dataset/images/test/00 ({img}).jpg'
    results = model(img)
    results.show()
```

j) S

k)