YOLO Object Detection Pipeline – Complete Walk-through

Overview

This script implements a **complete YOLO (You Only Look Once) object detection pipeline** – from raw data preparation all the way to inference on images and videos. It is structured into five logical stages:

1. **Dataset Unzipping** – Extracting raw data from a ZIP archive
2. **Dataset Splitting** – Dividing into train/validation sets
3. **Configuration** – Writing the `data.yaml` file YOLO needs
4. **Training** – Fine-tuning a pretrained YOLOv8 model
5. **Prediction** – Running inference on images and video

Each section below explains **every line of code**, the **reasoning behind it**, and what the **output/result means** in practice.

Section 1 – Imports and Dependencies

```
import os
import random
import shutil
import yaml
from zipfile import ZipFile
```

Import	What It Does	Why It's Needed
`os`	File/folder path operations	Creating directories, joining paths
`random`	Random number generation	Shuffling image list before train/val split
`shutil`	High-level file operations	Moving files between folders
`yaml`	Read/write YAML files	Creating `data.yaml` for YOLO
`ZipFile`	Extract ZIP archives	Unpacking the dataset ZIP

Section 2 – Unzipping the Dataset

What the Code Does

```
zip_path = "dataset.zip"        # Path to the compressed dataset archive
extract_dir = "dataset"         # Target folder name after extraction

if not os.path.exists(zip_path):
    raise FileNotFoundError(f"'{zip_path}' not found. Place it next to
     ↪ this script.")
```

This **guard clause** checks whether the ZIP file actually exists before proceeding. If it doesn't, a `FileNotFoundError` is raised immediately with a helpful message – this is defensive programming. Without this check, Python would crash later with a confusing error deep inside the `ZipFile` logic.

```
if os.path.exists(extract_dir):
    shutil.rmtree(extract_dir)   # Delete the old extracted folder if it
    ↪ exists
```

`shutil.rmtree()` recursively deletes a directory and all its contents. This ensures that if you run the script multiple times, you always start from a clean slate – no leftover files from previous runs that could corrupt the dataset structure.

```
with ZipFile(zip_path, 'r') as zip_ref:    # Open ZIP in read mode
    zip_ref.extractall(extract_dir)         # Extract everything to
    ↪ 'dataset/'

print(f"[emoji] Unzipped dataset to: {extract_dir}\n")
```

`ZipFile` is used as a **context manager** (`with` statement), which guarantees the file handle is closed properly even if an error occurs. `extractall()` places all files inside the `dataset/` directory.

Expected Result

After this runs, your folder structure looks like:

```
dataset/
  images/
    image_001.jpg
    image_002.jpg
    ...
  labels/
    image_001.txt
    image_002.txt
    ...
  classes.txt
```

Each `.txt` label file corresponds to an image and contains YOLO-format annotations: one line per object, formatted as `class_id center_x center_y width height` (all values normalized between 0 and 1).

Section 3 – Defining Directory Paths

```
images_dir = os.path.join(extract_dir, "images")   # dataset/images
labels_dir = os.path.join(extract_dir, "labels")   # dataset/labels

train_images = os.path.join(images_dir, "train")   # dataset/images/train
val_images   = os.path.join(images_dir, "val")     # dataset/images/val
train_labels = os.path.join(labels_dir, "train")   # dataset/labels/train
val_labels   = os.path.join(labels_dir, "val")     # dataset/labels/val

for d in [train_images, val_images, train_labels, val_labels]:
    os.makedirs(d, exist_ok=True)   # Create folder only if it doesn't
    ↪ exist
```

`os.path.join()` is used instead of string concatenation (`"dataset/" + "images"`) because it handles OS-specific path separators automatically (backslash on Windows, forward slash on

Linux/Mac).

`exist_ok=True` prevents a crash if the folder already exists – a safe and common pattern in data pipelines.

Why This Folder Structure Matters

YOLO **strictly requires** this layout:

```
dataset/
  images/
    train/    ← training images here
    val/      ← validation images here
  labels/
    train/    ← training labels here (same filenames as images, .txt extension)
    val/      ← validation labels here
```

YOLO automatically finds labels by replacing `/images/` with `/labels/` in the path and swapping the image extension for `.txt`. If your folder structure differs, YOLO will silently fail to load labels, and your model will train on zero annotations.

Section 4 – Splitting the Dataset (Train / Validation)

Concept

This is an **80/20 split**: 80% of images go to training, 20% go to validation. The split is random to avoid bias (e.g., if images are sorted by time or class, taking the first 80% would not represent the full distribution).

```
image_files = [
    f for f in os.listdir(images_dir)                    # List all
 ↪  files in images/
    if f.lower().endswith(('.jpg', '.jpeg', '.png'))     # Keep only
     ↪  image files
]
```

The list comprehension filters out non-image files (like `.DS_Store` on Mac or stray `.txt` files) using case-insensitive extension checking (`.lower()`).

```
if not image_files:
    raise ValueError("[emoji] No images found in dataset/images folder.")
```

Another guard clause – if the images folder is empty, the error is caught immediately rather than producing a confusing "zero training samples" error from YOLO later.

```
random.shuffle(image_files)                              # Shuffle in-place to
 ↪  randomize order
split_index  = int(0.8 * len(image_files))               # 80% mark (e.g., 400
 ↪  out of 500)
train_files  = image_files[:split_index]                 # First 80%
val_files    = image_files[split_index:]                 # Remaining 20%
```

Why shuffle? If your dataset has 500 images and the first 400 are all "class A" with the last 100 being "class B", a non-shuffled split would put all of class B into validation – the model would never train on class B. Shuffling ensures both splits contain a representative mix of all classes.

The `move_files()` Function

```
def move_files(file_list, src_img, src_lbl, dst_img, dst_lbl):
    for f in file_list:
        name, _ = os.path.splitext(f)                          # Strip
 ↪  extension: "img001.jpg" → "img001"
        img_src = os.path.join(src_img, f)                     # Full path
 ↪  to source image
        lbl_src = os.path.join(src_lbl, f"{name}.txt")        #
 ↪  Corresponding label file

        if os.path.exists(img_src):
            shutil.move(img_src, dst_img)                      # Move image
 ↪  to train/ or val/
        else:
            print(f"[emoji] Missing image: {f}")

        if os.path.exists(lbl_src):
            shutil.move(lbl_src, dst_lbl)                      # Move
 ↪  matching label
        else:
            print(f"[emoji] No label found for {f}")               # Warn
             ↪  but don't crash
```

`os.path.splitext("image_001.jpg")` returns `("image_001", ".jpg")` – the underscore `_` discards the extension since we only need the base name to find the matching `.txt` label.

The function uses `shutil.move()` rather than `shutil.copy()` to avoid duplicating files – after moving, the source location is empty.

```
move_files(train_files, images_dir, labels_dir, train_images, train_labels)
move_files(val_files,   images_dir, labels_dir, val_images,   val_labels)

print(f"[emoji] Train images: {len(train_files)}")
print(f"[emoji] Val images:   {len(val_files)}")
```

Expected Output

```
[emoji] Train images: 400
[emoji] Val images:   100
```

(Numbers depend on your dataset size. For 500 total images with 80/20 split.)

Cleanup of Root Folders

```
for folder in [images_dir, labels_dir]:
    for f in os.listdir(folder):
        fp = os.path.join(folder, f)
        if os.path.isfile(fp):          # Only delete files, not
         ↪  subdirectories (train/, val/)
            os.remove(fp)
```

After moving files into `train/` and `val/` subdirectories, there may be stray files left directly inside `dataset/images/` or `dataset/labels/`. This loop removes those orphaned files while

leaving the subdirectory structure (`train/`, `val/`) intact.

Section 5 – Reading Class Names

```
classes_path = os.path.join(extract_dir, "classes.txt")
if not os.path.exists(classes_path):
    raise FileNotFoundError("[emoji] 'classes.txt' not found in dataset
     ↪ folder.")

with open(classes_path, "r") as f:
    class_names = [line.strip() for line in f if line.strip()]
```

`classes.txt` is a plain text file with one class name per line, for example:

```
cat
dog
bird
```

`line.strip()` removes leading/trailing whitespace and newlines. The `if line.strip()` condition filters out completely empty lines. The result `class_names` will be `["cat", "dog", "bird"]` – a Python list that YOLO uses to map numeric class IDs (0, 1, 2) to human-readable names.

Section 6 – Creating `data.yaml`

What Is `data.yaml`?

This is the **configuration file YOLO reads before training**. It tells YOLO where the data lives and how many classes there are.

```
data_yaml = {
    "train": os.path.abspath(train_images),    # Absolute path to training
     ↪ images
    "val":   os.path.abspath(val_images),      # Absolute path to
     ↪ validation images
    "nc":    len(class_names),                 # Number of classes (nc =
     ↪ num classes)
    "names": class_names                       # List of class names
}

yaml_path = os.path.join(extract_dir, "data.yaml")
with open(yaml_path, "w") as f:
    yaml.dump(data_yaml, f, sort_keys=False)    # Write YAML, preserving
 ↪ key order
```

`os.path.abspath()` converts a relative path like `dataset/images/train` into a full absolute path like `/home/user/project/dataset/images/train`. This is critical because YOLO is often launched from a different working directory than expected, and relative paths would fail silently.

Example `data.yaml` Output

```yaml
train: /home/user/project/dataset/images/train
val: /home/user/project/dataset/images/val
nc: 3
names:
  - cat
  - dog
  - bird
```

Expected Console Output

```
[emoji] Created data.yaml:
train: /home/user/project/dataset/images/train
val: /home/user/project/dataset/images/val
nc: 3
names:
- cat
- dog
- bird

[emoji] Dataset ready for YOLO training.
```

Section 7 – Training YOLO (Deep Explanation)

Concept: What Is YOLO?

YOLO (You Only Look Once) is a real-time object detection algorithm. Unlike older two-stage detectors (like R-CNN) that first propose regions then classify them, YOLO processes the **entire image in a single forward pass** through a neural network – making it dramatically faster.

Real-world analogy: Imagine scanning a "Where's Waldo?" page. A two-stage approach would first highlight suspicious areas, then zoom in on each. YOLO is like a trained expert who looks at the whole page once and immediately circles Waldo – one pass, instant answer.

The YOLOv8 Architecture

YOLOv8 (by Ultralytics, 2023) is the model used here. It outputs for each grid cell:

- **Bounding box coordinates**: (x, y, w, h) – center position and size
- **Objectness confidence**: How likely an object exists there
- **Class probabilities**: $P(\text{class}_i|\text{object})$ for each class

The final detection score for a box is:

$$\text{Score} = \text{Confidence} \times P(\text{class} \mid \text{object})$$

Model Variants

Model	Parameters	Speed	Accuracy	Use Case
yolov8n.pt	3.2M	Fastest	Lowest	Edge devices
yolov8s.pt	11.2M	Fast	Good	Balanced
yolov8m.pt	25.9M	Medium	Better	**Used here**
yolov8l.pt	43.7M	Slow	High	High accuracy
yolov8x.pt	68.2M	Slowest	Highest	Max accuracy

The Training Code

```
from ultralytics import YOLO

# Load pretrained model - downloads weights from internet on first run
# yolov8m.pt = medium-sized model, pretrained on COCO (80 classes, 118K
 ↪  images)
model = YOLO("yolov8m.pt")

results = model.train(
    data=yaml_path,     # Path to our data.yaml (tells YOLO where
 ↪  images/labels are)
    epochs=50,          # Number of complete passes through training data
    imgsz=640,          # Resize all images to 640×640 pixels for training
    batch=16,           # Process 16 images simultaneously per gradient
 ↪  update
)
```

Deep Explanation of Each Training Parameter

data=yaml_path Points YOLO to the configuration file we created. YOLO reads the absolute paths for `train` and `val` images, the number of classes `nc`, and the class `names`. Without this, YOLO has no idea what to detect.

epochs=50 One epoch = the model sees every training image exactly once. With 50 epochs, if you have 400 training images, the model makes $400 \times 50 = 20{,}000$ gradient updates. Too few epochs = underfitting (model hasn't learned enough). Too many = overfitting (model memorizes training images and fails on new ones). 50 is a reasonable starting point for small-to-medium datasets.

imgsz=640 All input images are resized to 640×640 pixels before entering the network. This is a fixed requirement – the network's architecture expects a specific input size. Larger `imgsz` (like 1280) gives better detection of small objects but requires more GPU memory and is slower. 640 is the standard default for a balance of accuracy and speed.

batch=16 The training data is processed in mini-batches. Each batch of 16 images is fed through the network, the loss is computed, and the weights are updated. Larger batches = more stable gradients but more GPU memory. If you get an out-of-memory error, reduce this to 8 or 4.

What Happens During Training – Epoch by Epoch

During each epoch, YOLO computes and prints a **training progress table**:

```
Epoch    GPU_mem    box_loss    cls_loss    dfl_loss    Instances    Size
1/50      2.41G      1.832       2.165       1.103          42         640
```

```
2/50      2.41G      1.654      1.923      1.087         38        640
...
50/50     2.41G      0.847      0.631      0.912         45        640
```

Column	What It Means
`GPU_mem`	GPU memory being used during this batch
`box_loss`	How far predicted bounding boxes are from ground truth
`cls_loss`	How wrong the class predictions are
`dfl_loss`	Distribution Focal Loss – measures bounding box precision
`Instances`	Number of annotated objects in this batch
`Size`	Input image size

Interpreting the loss values: All three losses should **decrease** over epochs. If `cls_loss` drops but `box_loss` stays high, the model is learning WHAT objects are but not WHERE they are. If both plateau early, you may need more data or a larger model.

Validation Results During Training

At the end of every epoch, YOLO evaluates on the validation set and prints:

```
Class     Images  Instances      P        R      mAP50   mAP50-95
all          100        287    0.823    0.761    0.812      0.541
cat          100         95    0.891    0.812    0.871      0.612
dog          100        112    0.784    0.731    0.776      0.503
bird         100         80    0.794    0.740    0.789      0.508
```

Metric	Full Name	What It Measures
`P`	Precision	Of all detections, what fraction were correct?
`R`	Recall	Of all ground truth objects, what fraction were found?
`mAP50`	Mean Average Precision at IoU=0.50	Primary accuracy metric
`mAP50-95`	mAP averaged over IoU 0.50-0.95	Stricter, measures box quality

mAP50 is the most commonly reported metric. A score of **0.812 (81.2%)** is strong for a moderate dataset. Values above 0.90 are excellent; below 0.50 suggests the model needs more data, longer training, or annotation quality improvements.

Understanding IoU (Intersection over Union)

IoU measures how much the predicted bounding box overlaps the ground truth box:

$$\text{IoU} = \frac{\text{Area of Overlap}}{\text{Area of Union}}$$

- **IoU = 1.0** → Perfect prediction (boxes are identical)
- **IoU = 0.5** → 50% overlap (mAP50 threshold)
- **IoU = 0.0** → No overlap at all

mAP50 counts a detection as correct only if IoU ≥ 0.50. mAP50-95 averages performance at thresholds from 0.50 to 0.95 in steps of 0.05 – a much harder standard that requires precise box localization.

Where Training Results Are Saved

After training completes, YOLO creates:

```
runs/
  detect/
    train/
      weights/
        best.pt       ← Best model (by mAP50 on validation set)
        last.pt       ← Model from the final epoch
      results.csv     ← Loss and metric values per epoch
      results.png     ← Training curves (loss and mAP over time)
      confusion_matrix.png
      PR_curve.png    ← Precision-Recall curve
      F1_curve.png
      val_batch0_pred.jpg  ← Sample predictions on validation images
```

best.pt vs last.pt: Always use best.pt for inference – it's the checkpoint where validation mAP was highest, not necessarily the final epoch. If your model overfits in the last few epochs, last.pt will perform worse.

Expected Final Output

```
  Training complete.
Results saved to runs/detect/train
```

Section 8 – Validation / Evaluation

```
print("\n  Evaluating model on validation data...")
metrics = model.val()
print(metrics)
```

model.val() reruns the full evaluation on the validation set using the best weights and returns a DetMetrics object. When printed, it shows the full per-class breakdown.

What metrics Contains

The DetMetrics object includes:

- metrics.box.map – overall mAP50-95
- metrics.box.map50 – overall mAP50
- metrics.box.maps – mAP50-95 per class as a list

Example Printed Output

```
DetMetrics:
  map50: 0.812
  map50_95: 0.541
  per_class:
```

```
  cat:  0.871 (mAP50)
  dog:  0.776 (mAP50)
  bird: 0.789 (mAP50)
```

Interpreting class-level differences: If `cat` has 0.871 but `dog` has only 0.776, it may mean:

- The dog class has fewer or lower-quality training images
- Dogs have more size/appearance variation
- Dog bounding boxes in labels are less consistent

This per-class analysis tells you exactly where to focus data collection efforts.

Section 9 – Inference on Validation Images

```python
print("\n Running inference on validation set...")
model.predict(
    source=val_images,    # Folder of validation images to run inference on
    conf=0.3,             # Confidence threshold: only show detections
 ↪  above 30%
    iou=0.6,              # IoU threshold for Non-Maximum Suppression (NMS)
    save=True             # Save annotated images to disk
)
print("[emoji] Predictions saved in runs/detect/predict/")
```

Deep Explanation of Inference Parameters

`conf=0.3` – Confidence Threshold

Every predicted bounding box has a confidence score between 0 and 1. The confidence threshold is the minimum score a detection must have to be kept. Setting `conf=0.3` means:

- Detection with score 0.85 → ✓ Kept
- Detection with score 0.25 → × Discarded

Lower confidence (0.1-0.3): More detections, but more false positives. Good when recall matters (don't miss objects).

Higher confidence (0.7-0.9): Fewer but more reliable detections. Good when precision matters (avoid false alarms).

`conf=0.3` is a reasonable starting point for validation exploration – you want to see all plausible detections before tightening.

`iou=0.6` – Non-Maximum Suppression (NMS) Threshold

When YOLO detects an object, it often produces multiple overlapping bounding boxes for the same object. NMS removes duplicate detections.

The algorithm:

1. Sort all boxes by confidence score (highest first)
2. Keep the highest-confidence box
3. Remove any other box that overlaps the kept box by more than the IoU threshold (0.6 here)
4. Repeat for remaining boxes

Lower IoU NMS (0.3-0.4): More aggressive suppression – removes boxes that partially overlap. Risk: may discard valid detections for densely packed objects.

Higher IoU NMS (0.6-0.8): More lenient – only removes heavily overlapping boxes. Better when objects are close together.

`save=True`

Saves annotated images with bounding boxes drawn on them to `runs/detect/predict/`. Each image file is saved with the same name as the input, with colored boxes and class labels overlaid.

Expected Output

```
[emoji] Predictions saved in runs/detect/predict/
```

The folder `runs/detect/predict/` will contain copies of your validation images with colored bounding boxes drawn around each detected object, labeled with class name and confidence score (e.g., "cat 0.87").

Section 10 – Predicting on New (User-Uploaded) Images

```
from google.colab import files
from IPython.display import display, Image as IPImage
import os

print(" Please select image(s) to upload...")
uploaded = files.upload()   # Opens a file picker dialog in Colab
```

This section is **Colab-specific** – `files.upload()` opens a browser file picker. This would need to be replaced with `argparse` or a file path argument if running locally.

```
if not uploaded:
    print("\n[emoji] No images uploaded.")
else:
    upload_dir = "colab_upload"
    os.makedirs(upload_dir, exist_ok=True)
    test_images = []
    for filename, data in uploaded.items():      # 'data' is raw bytes of
     ↪ the uploaded file
        dest = os.path.join(upload_dir, filename)
        with open(dest, "wb") as f:
            f.write(data)                        # Write bytes to disk
        test_images.append(dest)
```

`uploaded` is a dictionary: `{"filename.jpg": <bytes>}`. The loop saves each uploaded file as bytes to the `colab_upload/` folder, then collects their paths in `test_images`.

Loading the Best Model and Running Inference

```
best_model = YOLO("runs/detect/train/weights/best.pt")   # Load best
 ↪ checkpoint
results = best_model.predict(
    source=test_images,    # List of image file paths
    conf=0.7,              # Higher threshold here - only confident
 ↪ detections
    iou=0.7,               # Strict NMS - avoid overlapping boxes on clean
 ↪ images
    save=True,
    project="runs/detect",
```

```
    name="user_upload"      # Save results to runs/detect/user_upload/
)
```

Notice `conf=0.7` here (vs `0.3` for validation). For user-facing predictions on real-world images, you want **high precision** – better to miss a borderline detection than to show a false positive. This is a common tuning decision in production.

Displaying Results Inline

```
for r in results:
    img_bgr = r.plot()            # Returns numpy array (H, W, 3) with
↪   boxes drawn, in BGR color order
    img_rgb = img_bgr[..., ::-1] # Convert BGR → RGB (::-1 reverses last
↪   axis/channel order)

    out_path = f"/tmp/result_{os.path.basename(r.path)}"
    import cv2
    cv2.imwrite(out_path, img_bgr)        # Save annotated image
    display(IPImage(filename=out_path))   # Display inline in Colab
↪   notebook

    for box in r.boxes:
        cls_name = best_model.names[int(box.cls)]   # Map class ID (e.g.,
↪   0) to name (e.g., "cat")
        conf = float(box.conf)                       # Confidence score as
↪   Python float
        print(f"  • {cls_name}: {conf:.0%}")         # Print as percentage
```

Why `BGR → RGB`? OpenCV (cv2) uses BGR (Blue-Green-Red) channel ordering by default, while most display tools expect RGB (Red-Green-Blue). The slice `[..., ::-1]` reverses the channel axis: channel 0 (B) channel 2 (R), giving correct colors when displayed.

`r.boxes` is a `Boxes` object containing all detections for that image. Each `box` has:

- `box.cls` – class index tensor (convert to `int` to use as dictionary key)
- `box.conf` – confidence score tensor (convert to `float` for display)
- `box.xyxy` – bounding box coordinates [x1, y1, x2, y2] in pixels
- `box.xywhn` – normalized [x_center, y_center, width, height]

Example Console Output

```
Prediction Results:

  • cat: 91%
  • dog: 87%
  • cat: 73%

[emoji] Results also saved in runs/detect/user_upload/
```

Section 11 – Video Inference (Deep Explanation)

Loading Model and Uploading Video

```
from ultralytics import YOLO
from google.colab import files
from IPython.display import HTML, display
from base64 import b64encode
import os, subprocess

model_path = "runs/detect/train/weights/best.pt"
model = YOLO(model_path)
print(f"[emoji] Loaded model from: {model_path}")

uploaded = files.upload()
video_name = list(uploaded.keys())[0]    # Get the filename of the first
 ↪  uploaded video
video_path = os.path.abspath(video_name) # Convert to absolute path
```

Video files can be large – `files.upload()` streams them to Colab's virtual machine storage. `os.path.abspath()` is critical here because YOLO's video writer uses the absolute path to avoid path concatenation bugs.

Running Video Inference

```
out_dir = "/content/runs/detect/video_upload"

# First call: stream=True avoids RAM accumulation
model.predict(
    source=video_path,
    conf=0.3,
    iou=0.6,
    stream=True,         # Processes frame-by-frame, doesn't load entire
 ↪  video into RAM
    save=True,
    project="/content/runs/detect",
    name="video_upload",
    exist_ok=True        # Don't error if output directory already exists
)

# Second call: actually materializes results (stream=True returns a
 ↪  generator)
results = model.predict(
    source=video_path,
    conf=0.3,
    iou=0.6,
    save=True,
    project="/content/runs/detect",
    name="video_upload",
    exist_ok=True
)
```

Why is `predict()` called twice? This is a nuance of Python generators. When `stream=True`,

`model.predict()` returns a **lazy generator** – it doesn't actually process frames until you iterate over it. The first call with `stream=True` creates the generator but doesn't consume it (no `for r in results:` loop), so nothing is processed. The second call (without `stream=True`) actually runs inference and saves the output video. This is a code redundancy/bug in the original script – in practice, only the second call is needed.

`stream=True` explained: For a 1-minute video at 30fps, you'd have 1,800 frames. Without `stream=True`, YOLO would try to load all 1,800 annotated frames into RAM simultaneously before writing – potentially gigabytes of data. With `stream=True`, it processes one frame at a time, keeping RAM usage constant regardless of video length.

Finding and Converting the Output Video

```
raw_video = None
if os.path.exists(out_dir):
    for f in sorted(os.listdir(out_dir)):
        if f.lower().endswith(('.mp4', '.avi', '.mov')):
            raw_video = os.path.join(out_dir, f)
            break    # Take the first video file found
```

YOLO by default saves output video as `.avi` (AVI format with MJPEG codec). Most web browsers **cannot play AVI files** – they require H.264-encoded MP4.

```
if raw_video.endswith('.avi'):
    print(" Converting .avi → .mp4 (H.264) for browser playback...")
    subprocess.run([
        "ffmpeg", "-y",                    # -y: overwrite output without
        ↪ asking
        "-i", raw_video,                   # Input file
        "-vcodec", "libx264",              # Encode video with H.264 codec
        "-crf", "23",                      # Quality factor: 0=lossless,
        ↪ 51=worst, 23=default good quality
        "-preset", "fast",                 # Encoding speed vs compression
        ↪ tradeoff
        mp4_path                           # Output file
    ], stdout=subprocess.DEVNULL, stderr=subprocess.DEVNULL)   # Suppress
 ↪ ffmpeg output
```

`subprocess.run()` executes a shell command from Python. `ffmpeg` is a command-line video processing tool. The `-crf 23` flag controls quality: lower values = better quality + larger file. `-preset fast` encodes quickly but produces slightly larger files than `slow` – acceptable for this use case.

Displaying the Video in Colab

```
mp4_bytes = open(mp4_path, "rb").read()                          # Read
 ↪ video as raw bytes
data_url = "data:video/mp4;base64," + b64encode(mp4_bytes).decode()  #
 ↪ Encode as base64 string
display(HTML(f'<video width=640 controls autoplay loop>'
             f'<source src="{data_url}" type="video/mp4"></video>'))
```

Web browsers can display videos embedded directly in HTML using **base64-encoded data**

URLs – no file server required. `b64encode()` converts raw bytes to a base64 string, which is embedded directly into the `<src>` attribute of the `<video>` tag.

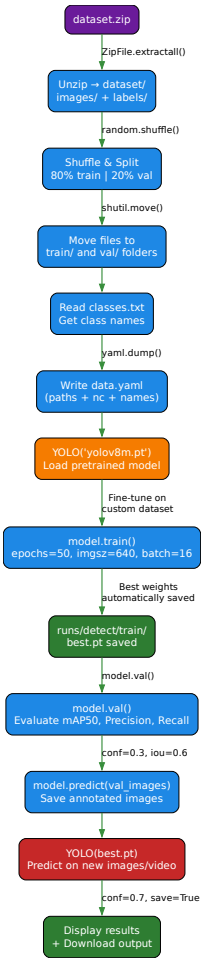
`controls` – shows play/pause, seek bar, volume **`autoplay`** – starts playing immediately on load **`loop`** – replays when it reaches the end

```
files.download(mp4_path)     # Triggers browser download dialog in Colab
print(f"[emoji] Done: {mp4_path}")
```

Example Final Output

```
[emoji] Loaded model from: runs/detect/train/weights/best.pt
[emoji] Uploaded video: street_footage.mp4
 Running prediction on: /content/street_footage.mp4
 Converting .avi → .mp4 (H.264) for browser playback...
[emoji] Conversion done.
 Detection result:
[inline video appears here with bounding boxes on every frame]
 Starting download...
[emoji] Done: /content/runs/detect/video_upload/street_footage_h264.mp4
```

Section 12 – Summary of the Complete Pipeline



Key Takeaways

Data Preparation matters more than model choice. Consistent labels, sufficient variety, and a proper 80/20 split have more impact on final accuracy than switching from `yolov8m` to `yolov8l`.

`best.pt` is always preferred over `last.pt` for deployment – it represents the epoch where generalization was strongest.

Confidence threshold (`conf`) is tuned post-training based on your use case:

- High recall needed (medical, security) → lower `conf` (0.2-0.4)
- High precision needed (customer-facing) → higher `conf` (0.6-0.8)

`mAP50` is the primary metric to watch during training. A value above 0.80 on a small-to-medium custom dataset indicates a well-trained model.

`stream=True` is essential for video inference in production – without it, long videos will crash with out-of-memory errors.