Research

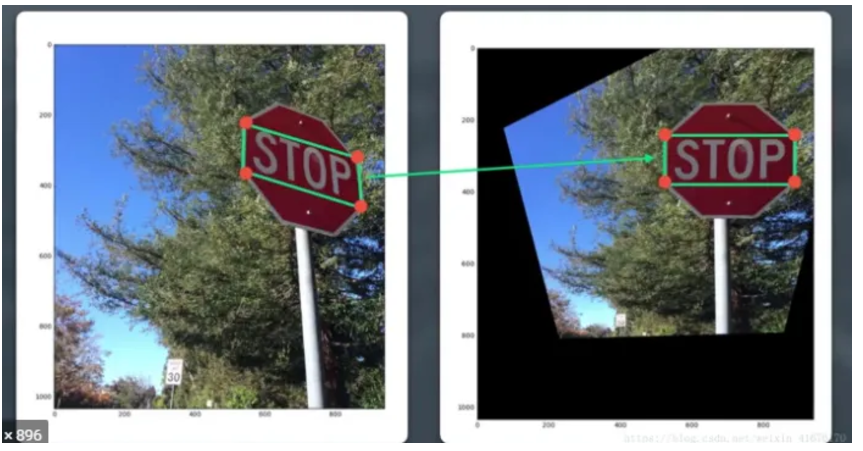
Data augmentation

* Perspective

Perspective transform augmentation involves distorting the image to simulate perspective changes. This is particularly useful for scenarios where objects may appear at different distances or viewpoints.

By applying perspective transformations during training, the YOLO model learns to handle variations in object sizes, shapes, and distortions caused by perspective changes.

perspective: 0.0 # image perspective (+/- fraction), range 0-0.001



* Image scale
* Check image mosaic
* Cut mix

Default Augmentation Flags in YOLOv8

Flag Default Value Description

hsv\_h 0.015 Default range for random hue adjustments in HSV space.

hsv\_s 0.7 Default range for random saturation adjustments in HSV space.

hsv\_v 0.4 Default range for random value (brightness) adjustments in HSV space.

flipud 0.0 Default probability of vertical flipping (disabled by default).

fliplr 0.5 Default probability of horizontal flipping.

degrees 0.0 Default range for random rotation (disabled by default).

translate 0.1 Default range for random translation along X and Y axes.

scale 0.5 Default range for random scaling (zoom in or out).

shear 0.0 Default range for random shearing (disabled by default).

perspective 0.0 Default level of perspective distortion (disabled by default).

mosaic 1.0 Mosaic augmentation enabled by default (applied to all images).

mixup 0.0 Mixup augmentation disabled by default.

blur 0.0 Default level of Gaussian blur (disabled by default).

rect False Rectangular training disabled by default.

augment True General augmentation enabled by default.

* Class specific training - (expand the stable diffusion thing)
* Look over hyperparameters
* 4. Freeze Layers
* When using a pretrained backbone, you can freeze the initial layers for the first few epochs to allow the detection head to adapt first.
* Example Code:
* python
* Copy code
* from ultralytics import YOLO
* # Load the YOLOv8x model
* model = YOLO('yolov8x.pt')
* # Freeze layers
* model.train(
* data='data.yaml',
* epochs=100,
* imgsz=640,
* freeze=10 # Freeze first 10 layers
* )
* Alternatively, from the command line:
* bash
* Copy code
* yolo train model=yolov8x.pt data=data.yaml epochs=100 imgsz=640 freeze=10

4. Progressive Training

Train on lower resolution (e.g., 320x320) images initially for faster convergence.

Fine-tune at higher resolution (e.g., 640x640) to improve precision.

5. Synthetic Data or Bootstrapping

Use techniques like:

Generating synthetic images with tools like roboflow or albumentations.

Adding external data for the underrepresented class.

Class imbalance mitigation can be effectively handled in YOLOv8 through two primary techniques: weighted loss functions and balanced data sampling. Here's how you can implement both approaches:

1. Weighted Loss Functions

YOLOv8 allows you to specify weights for each class in the dataset. This adjusts the loss computation to prioritize underrepresented classes by assigning them higher weights.

Steps:

Prepare Class Weights:

Calculate the weight for each class based on the inverse frequency of its occurrences in the dataset.

Example formula for weights:

weight

𝑖

=

1

frequency

𝑖

weight

i

​

=

frequency

i

​

1

​

Normalize the weights so they sum to 1.

Modify Training Configuration:

Update the class\_weights field in the training script.

Example Python code to set class weights:

python

Copy code

from ultralytics import YOLO

# Define class weights (replace with your calculated weights)

class\_weights = [2.0, 1.0, 0.5] # Higher weight for underrepresented classes

# Load the model

model = YOLO("yolov8n.pt")

# Train with class weights

model.train(

data="data.yaml",

epochs=50,

imgsz=640,

class\_weights=class\_weights

)

2. Balanced Data Sampling

YOLOv8 does not have built-in options for balanced sampling, but you can preprocess your dataset to achieve it. You can either over-sample minority classes or under-sample majority classes.

Option 1: Over-Sampling Minority Classes

Duplicate images containing minority classes to increase their frequency in the dataset.

Implementation:

Identify images with minority class annotations.

Duplicate these images and their corresponding labels to balance the dataset.

Example:

python

Copy code

import os

import shutil

# Define paths

images\_dir = "path/to/images"

labels\_dir = "path/to/labels"

# Target class to oversample (e.g., class 0)

target\_class = 0

for label\_file in os.listdir(labels\_dir):

if label\_file.endswith(".txt"):

with open(os.path.join(labels\_dir, label\_file), "r") as f:

lines = f.readlines()

# Check if target class exists in the label file

if any(line.startswith(f"{target\_class} ") for line in lines):

# Duplicate image and label

image\_file = label\_file.replace(".txt", ".jpg")

shutil.copy(os.path.join(images\_dir, image\_file), os.path.join(images\_dir, f"dup\_{image\_file}"))

shutil.copy(os.path.join(labels\_dir, label\_file), os.path.join(labels\_dir, f"dup\_{label\_file}"))

print("Over-sampling completed.")

Option 2: Under-Sampling Majority Classes

Reduce the number of images with majority classes to balance the dataset.

Implementation:

Identify images with majority class annotations.

Remove a subset of these images and their corresponding labels.

3. Combining Weighted Loss and Sampling

For optimal results, you can combine weighted loss and balanced data sampling:

Over-sample minority classes to ensure adequate representation.

Apply class weights to ensure the model focuses on underrepresented classes during training.

Best Practices

Monitor Model Performance: Track class-specific metrics (precision, recall, F1-score) to ensure improvements in underrepresented classes.

Avoid Overfitting: Ensure that over-sampling doesn’t lead to overfitting on minority classes.

Dynamic Adjustment: Update class weights periodically during training based on evolving class distributions.

By implementing these techniques, YOLOv8 can handle class imbalance effectively and deliver improved detection results for underrepresented classes.