Harvard University Extension School

ALM in Data Science

CSCI E-89 Deep Learning

REPORT

Crack Detection and localization in Civil Infrastructure Using YOLOv11

Seymur Hasanov

December 18, 2024

Executive Summary

This project focuses on developing an automated crack detection and classification system for civil infrastructure using the YOLOv11 object detection model. The primary objective is to identify and categorize cracks into 11 distinct classes, including diagonal, horizontal, vertical cracks, and tile damage, to assist in structural health monitoring.

A dataset comprising 2159 annotated images was utilized, divided into training (1505 images), validation (422 images), and test (232 images) sets. Preprocessing techniques such as grayscale conversion, histogram equalization, and data augmentation were applied to enhance model robustness and generalization.

Multiple YOLOv11 models, including YOLOv11n (nano) and YOLOv11x (extra-large), were trained and compared. Hyperparameter tuning, such as learning rate adjustment, optimizer selection, and weight decay regularization, was performed to stabilize training and improve performance.

The results showed that:

YOLOv11x achieved the best performance with a mean Average Precision (mAP50) of 70.7% and mAP50-95 of 49.5%, demonstrating superior generalization. Precision and recall improved steadily over training epochs, particularly for severe crack types. Some challenges, such as misclassification of certain fine and medium cracks, remain and highlight areas for further improvement. This project successfully demonstrates the potential of deep learning-based solutions for real-world structural health monitoring, offering a scalable and efficient tool to improve infrastructure safety and reliability for engineers and maintenance professionals.

Introduction

The growing need for reliable and automated crack detection systems is essential for maintaining the safety and longevity of civil infrastructure. This project focuses on developing an efficient deep learning-based solution to detect and classify cracks in structural elements such as pavements, walls, and tiles. Leveraging the YOLOv11 model, known for its real-time performance and accuracy, this work aims to identify crack types categorized into 11 distinct classes.

A dataset (Civil Faults) consisting of 2159 annotated images was downloaded from Roboflow and was utilized, with train, validation, and test splits prepared to ensure balanced model evaluation. Key preprocessing steps, including histogram equalization and image augmentation, were applied to enhance model robustness and performance. Hyperparameter tuning, such as learning rate adjustment and weight decay optimization, further improved detection accuracy.

By addressing the challenges of crack detection using state-of-the-art deep learning techniques, this project demonstrates a practical and scalable solution for structural health monitoring, contributing to safer infrastructure management.

Importing Libraries and Packages

Essential libraries are imported to facilitate data processing and model training. Ultralytics YOLOv11n and x models are utilized for building and training the object detection model, while NumPy and OpenCV handle numerical operations and image preprocessing. Matplotlib is used for visualizing results and metrics. These tools ensure an efficient setup for image handling and deep learning tasks.

```
[ ] !pip install roboflow
    !pip install -U albumentations
    %pip install "ultralytics<=8.3.40" supervision roboflow
```

```
import tensorflow as tf
import ultralytics
from tensorflow import keras
import os
import matplotlib.pyplot as plt
import cv2
import albumentations as A
from albumentations.pytorch import ToTensorV2
import matplotlib.pyplot as plt
import numpy as np
from PIL import Image
import glob
import pandas as pd
import seaborn as sns
from roboflow import Roboflow
```

Dataset Downloading and Overview

The dataset was downloaded from (Roboflow), a platform providing pre-processed and annotated datasets for machine learning tasks. Using the Roboflow API, the dataset was accessed with the following script:

```
[ ] rf = Roboflow(api_key="VJPVPFZzdmctDzTJre7S")
    project = rf.workspace("iiti").project("civil-faults-detection")
    version = project.version(1)
    dataset = version.download("yolov11")
```

The dataset contains 2159 images of cracks in civil structures, divided into three subsets:

- Training Set: 1505 images (70%)
- Validation Set: 422 images (20%)
- Test Set: 232 images (10%)

Each image is resized to 640x640 pixels, with auto-orientation applied during preprocessing. The dataset includes 11 classes of faults, such as diagonal, horizontal, vertical cracks, and tile damages. This structured and well-annotated dataset serves as the foundation for training and evaluating the YOLOv11 model for crack detection and classification.

The next cell counts the number of images in each subfolder (train, valid, test) and extracts unique class IDs from the label files. The images counts for each split are displayed, along with the list of detected class IDs to verify the dataset structure.

```
[ ] base_path = "./Civil-Faults-Detection--1"
    splits = ['train', 'valid', 'test']
    classes = set()

    # Function to count files in folders
    def count_images_and_classes(path):
        img_count = {}
        for split in splits:
            split_path = os.path.join(path, split, "images")
            label_path = os.path.join(path, split, "labels")

            # Count images
            num_images = len(os.listdir(split_path))
            img_count[split] = num_images

            # Gather class names from label files
            for file in os.listdir(label_path):
                with open(os.path.join(label_path, file), "r") as f:
                    lines = f.readlines()
                    for line in lines:
                        class_id = int(line.split()[0])
                        classes.add(class_id)

        return img_count
```

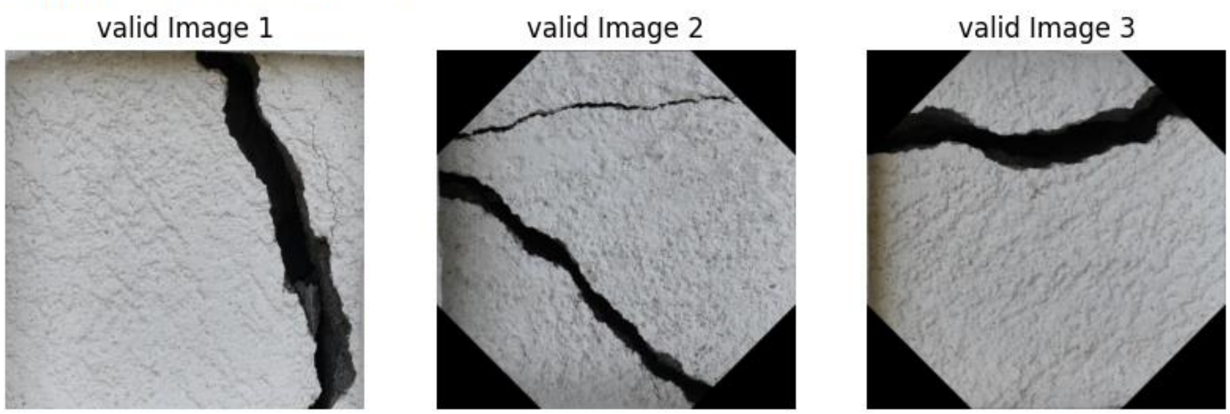
Image counts per split: {'train': 1505, 'valid': 422, 'test': 232}

Classes found: [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10]

The next cell displays sample images from the train, valid, and test sets to analyze their visual quality and diversity.

```
[ ]  # Function to display sample images
     def display_sample_images(path, split, num_images=3):
         img_folder = os.path.join(path, split, "images")
         images = os.listdir(img_folder)[:num_images]

         plt.figure(figsize=(10, 5))
         for i, img_name in enumerate(images):
             img_path = os.path.join(img_folder, img_name)
             img = cv2.imread(img_path)
             img = cv2.cvtColor(img, cv2.COLOR_BGR2RGB)

             plt.subplot(1, num_images, i+1)
             plt.imshow(img)
             plt.title(f"{split} Image {i+1}")
             plt.axis("off")
         plt.show()

     # Display sample images for train, valid, and test
     for split in splits:
         print(f"Displaying sample images from {split} set:")
         display_sample_images(base_path, split)
```


```
Displaying sample images from valid set:
```



```
Displaying sample images from test set:
```

Data Augmentations

Observations from the images include:

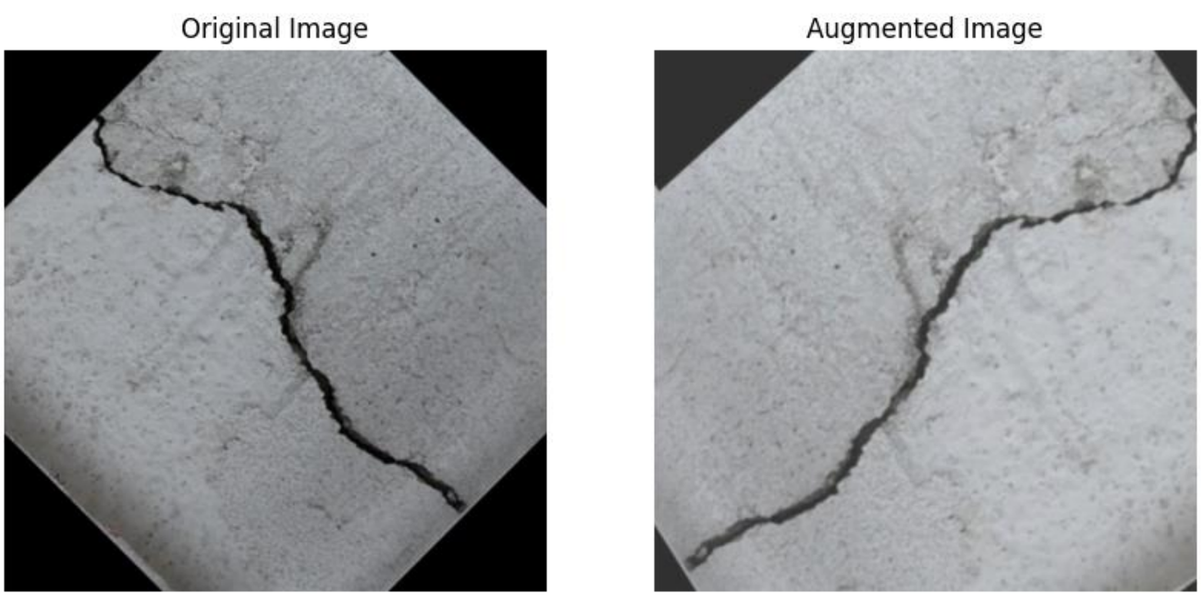
- **Shadows and Lighting Variations**: Uneven lighting and shadows can reduce the clarity of crack features.
- **Rotation and Alignment**: Some cracks appear at rotated angles, requiring alignment adjustments.
- **Background Noise**: The presence of grass, texture variations, and irregular surfaces adds complexity to the dataset.

To address these issues and enhance model robustness, data augmentation techniques are applied. The augmentations include *brightness/contrast adjustments, rotations, flips, cropping, blurring, noise addition, and resizing*. These transformations help simulate real-world variations and improve the model's generalization capabilities.

The next cell demonstrates these augmentations by applying them to a sample image, displaying the original and augmented versions for comparison.

```
# Define augmentations
augmentations = A.Compose([
    A.RandomBrightnessContrast(p=0.5),
    A.Rotate(limit=20, p=0.5),
    A.HorizontalFlip(p=0.5),
    A.VerticalFlip(p=0.5),
    A.RandomCrop(height=256, width=256, p=0.5),
    A.GaussianBlur(blur_limit=(3, 5), p=0.3),
    A.GaussNoise(var_limit=(10.0, 50.0), p=0.3),
    A.ColorJitter(p=0.5),
    A.Perspective(scale=(0.05, 0.1), p=0.5),
    A.Resize(640, 640),
    ToTensorV2()
])
```

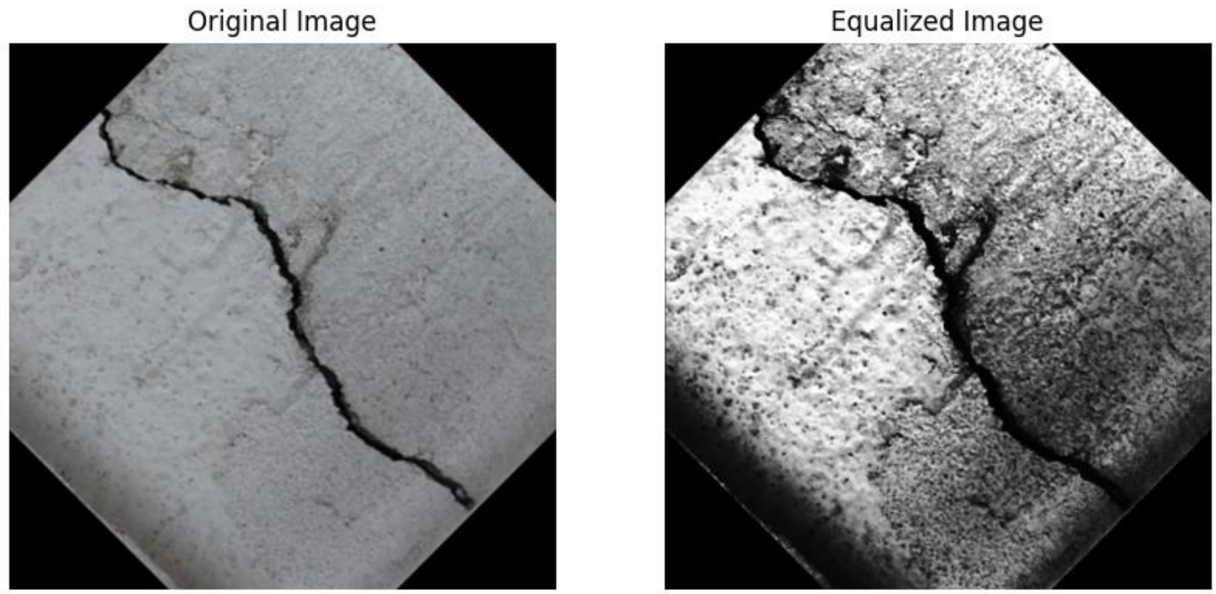
```
# Test augmentations on a sample image
image_path = "/content/Civil-Faults-Detection--1/train/images/DSC_3766rrr_jpg.rf.d21f2073bb298cfe62cb3b444d660394.jpg"
image = cv2.imread(image_path)
image = cv2.cvtColor(image, cv2.COLOR_BGR2RGB)

augmented = augmentations(image=image)["image"]

fig, axs = plt.subplots(1, 2, figsize=(10, 5))
axs[0].imshow(image)
axs[0].set_title("Original Image")
axs[0].axis("off")

axs[1].imshow(augmented.permute(1, 2, 0))
axs[1].set_title("Augmented Image")
axs[1].axis("off")

plt.show()
```



The augmented image shows slight rotation, a random crop focusing on a specific region, and adjustments to brightness and contrast. These transformations enhance image diversity, improving the model's ability to generalize to varying orientations, lighting conditions, and partial crack visibility.

Equalizations

The next step applies histogram equalization to enhance image contrast, which helps address shadows and uneven lighting conditions. By converting the image to grayscale and redistributing pixel intensities, this technique highlights crack features more clearly. The function displays the original image alongside the equalized image for visual comparison. This preprocessing step ensures that important details, such as cracks, are more distinguishable for the model.



Model Selection and Justification

The project uses the **YOLOv11n** model, a lightweight variant of the YOLOv11 family, specifically chosen for its balance between **accuracy** and **efficiency**. As shown in the table below, YOLOv11n achieves a respectable **mAP50-95 of 39.5** with a speed of **56.1 ms** on CPU and **1.5 ms** on T4 GPUs, while maintaining a small model size with **2.6 million parameters** and **6.5 GFLOPs**.

Why YOLOv11n?

- **Speed**: YOLOv11n provides real-time performance, making it ideal for practical applications such as crack detection in infrastructure.
- **Efficiency**: Its low computational cost allows deployment on systems with limited hardware resources.
- **Accuracy**: Despite being the smallest variant, YOLOv11n achieves competitive accuracy, making it suitable for the given task.

YOLOv11 Model Performance Comparison

Model	Size (pixels)	mAP50-95	Speed (CPU ONNX, ms)	Speed (T4 TensorRT10, ms)	Params (M)	FLOPs (B)
YOLO11n	640	39.5	56.1 ± 0.8	1.5 ± 0.0	2.6	6.5
YOLO11s	640	47.0	90.0 ± 1.2	2.5 ± 0.0	9.4	21.5
YOLO11m	640	51.5	183.2 ± 2.0	4.7 ± 0.1	20.1	68.0
YOLO11l	640	53.4	238.6 ± 1.4	6.2 ± 0.1	25.3	86.9
YOLO11x	640	54.7	462.8 ± 6.7	11.3 ± 0.2	56.9	194.9

Implementation Overview

The YOLOv11n model is loaded with pre-trained weights using the following configuration:

```
ultralytics.checks()
```

Show hidden output

Load the YOLOv11 model and set up the environment.

```
[ ] from ultralytics import YOLO
    model = YOLO("yolo11n.yaml").load("yolo11n.pt")  # build from YAML and transfer weights
```

```
Downloading https://github.com/ultralytics/assets/releases/download/v8.3.0/yolo11n.pt to 'yolo11n.pt'...
100%|██████████| 5.35M/5.35M [00:00<00:00, 65.3MB/s]Transferred 499/499 items from pretrained weights
```

Model Summary and Architecture

The YOLOv11n model consists of **319 layers**, **2.6 million parameters**, and requires **6.6 GFLOPs**, as shown in the following output:

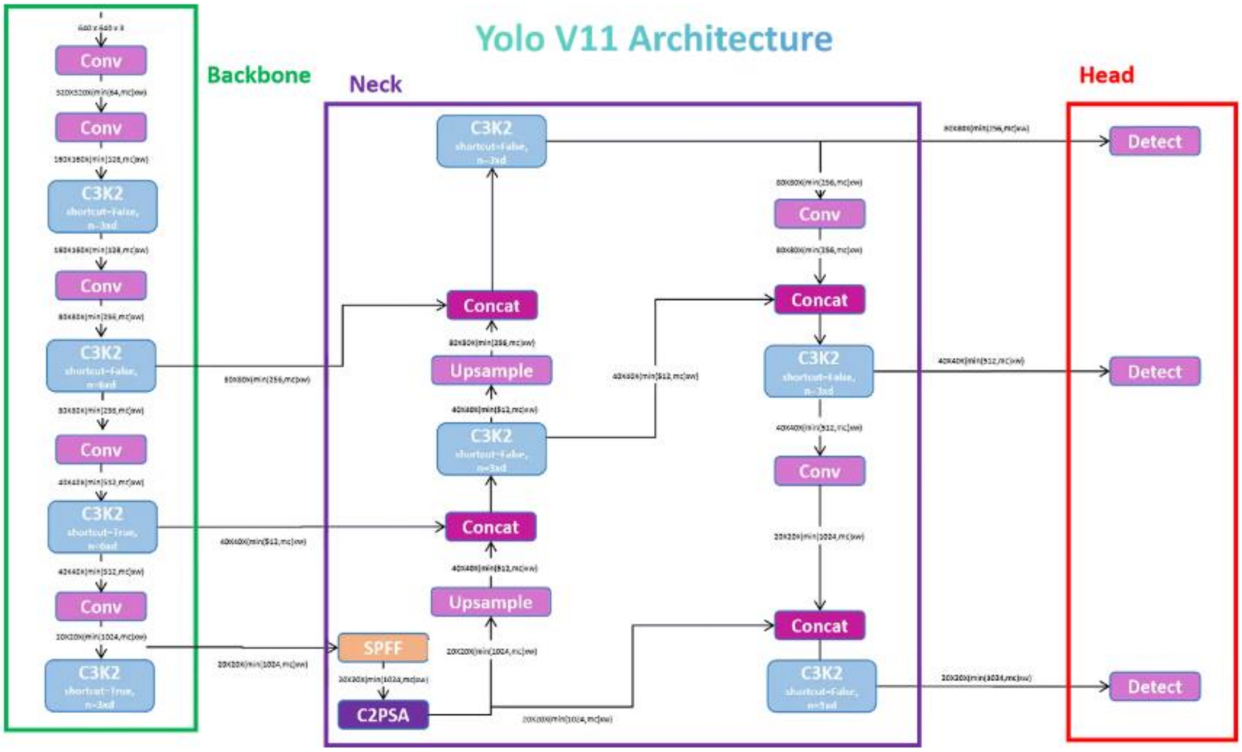
The architecture is divided into three main components:

1. **Backbone**: Extracts multi-scale features using **Conv** layers and **C3K2 blocks**.
2. **Neck**: Enhances feature fusion with **SPPF** and **C2PSA** blocks, combining upsampled feature maps.
3. **Head**: Detects and classifies objects at multiple scales using optimized **Detect** layers.

This design achieves a balance between computational efficiency and detection accuracy, making YOLOv11n ideal for real-time applications.

YOLOv11 Architecture

The figure below illustrates the [YOLOv11 architecture](), showing the **Backbone**, **Neck**, and **Head** components.




```
[ ] model.info()
```

```
YOLO11n summary: 319 layers, 2,624,080 parameters, 2,624,064 gradients, 6.6 GFLOPs
(319, 2624080, 2624064, 6.614336)
```

Model Training

The training process uses the following configurations:

- **Dataset**: /content/Civil-Faults-Detection--1/data.yaml
- **Epochs**: 50
- **Image Size**: 640x640 pixels
- **Optimizer**: AdamW (automatically determined)
- **Learning Rate**: Adjusted dynamically (initial lr0=0.01)
- **Batch Size**: 16
- **Patience**: 10 epochs for early stopping
- **Augmentations**: Built-in augmentations such as flipping, CLAHE, blur, and color jittering.

YAML File Overview

The **YAML file** provides the YOLO model with essential configuration details, including paths to the training, validation, and test datasets, the number of classes (nc), and class names. It ensures the model can locate the dataset and interpret the class labels correctly, enabling seamless training and evaluation.

The training command is as follows:

```
[ ] # Train the model
    results = model.train(
        data="/content/Civil-Faults-Detection--1/data.yaml",
        epochs=50,
        imgsz=640,
        plots=True,
        patience=10
    )
```

```
New https://pypi.org/project/ultralytics/8.3.51 available 😃 Update with 'pip install -U ultralytics'
Ultralytics 8.3.40 🚀 Python-3.10.12 torch-2.5.1+cu121 CUDA:0 (NVIDIA A100-SXM4-40GB, 40514MiB)
engine/trainer: task=detect, mode=train, model=yolo11n.yaml, data=/content/Civil-Faults-Detection--1/data.yaml,
Downloading https://ultralytics.com/assets/Arial.ttf to '/root/.config/Ultralytics/Arial.ttf'...
100%|██████████| 755k/755k [00:00<00:00, 14.9MB/s]
Overriding model.yaml nc=80 with nc=11

                   from  n    params  module                                       arguments
  0                  -1  1       464  ultralytics.nn.modules.conv.Conv             [3, 16, 3, 2]
  1                  -1  1      4672  ultralytics.nn.modules.conv.Conv             [16, 32, 3, 2]
  2                  -1  1      6640  ultralytics.nn.modules.block.C3k2            [32, 64, 1, False, 0.25]
  3                  -1  1     36992  ultralytics.nn.modules.conv.Conv             [64, 64, 3, 2]
  4                  -1  1     26080  ultralytics.nn.modules.block.C3k2            [64, 128, 1, False, 0.25]
  5                  -1  1    147712  ultralytics.nn.modules.conv.Conv             [128, 128, 3, 2]
  6                  -1  1     87040  ultralytics.nn.modules.block.C3k2            [128, 128, 1, True]
  7                  -1  1    295424  ultralytics.nn.modules.conv.Conv             [128, 256, 3, 2]
  8                  -1  1    346112  ultralytics.nn.modules.block.C3k2            [256, 256, 1, True]
  9                  -1  1    164608  ultralytics.nn.modules.block.SPPF            [256, 256, 5]
 10                  -1  1    249728  ultralytics.nn.modules.block.C2PSA           [256, 256, 1]
 11                  -1  1         0  torch.nn.modules.upsampling.Upsample         [None, 2, 'nearest']
 12             [-1, 6]  1         0  ultralytics.nn.modules.conv.Concat           [1]
 13                  -1  1    111296  ultralytics.nn.modules.block.C3k2            [384, 128, 1, False]
 14                  -1  1         0  torch.nn.modules.upsampling.Upsample         [None, 2, 'nearest']
 15             [-1, 4]  1         0  ultralytics.nn.modules.conv.Concat           [1]
 16                  -1  1     32096  ultralytics.nn.modules.block.C3k2            [256, 64, 1, False]
 17                  -1  1     36992  ultralytics.nn.modules.conv.Conv             [64, 64, 3, 2]
 18            [-1, 13]  1         0  ultralytics.nn.modules.conv.Concat           [1]
 19                  -1  1     86720  ultralytics.nn.modules.block.C3k2            [192, 128, 1, False]
 20                  -1  1    147712  ultralytics.nn.modules.conv.Conv             [128, 128, 3, 2]
 21            [-1, 10]  1         0  ultralytics.nn.modules.conv.Concat           [1]
 22                  -1  1    378880  ultralytics.nn.modules.block.C3k2            [384, 256, 1, True]
```

```
      Epoch    GPU_mem   box_loss   cls_loss   dfl_loss  Instances       Size
      49/50       2.6G     0.7751     0.8312      1.179          1        640: 100%|██████████| 95/95 [00:10<00:00,  9.07it/s]
                 Class     Images  Instances      Box(P          R       mAP50  mAP50-95): 100%|██████████| 14/14 [00:01<00:00,  8.30it/s]

      Epoch    GPU_mem   box_loss   cls_loss   dfl_loss  Instances       Size
      50/50      2.62G     0.7573     0.8017      1.161          2        640: 100%|██████████| 95/95 [00:10<00:00,  8.92it/s]
                 Class     Images  Instances      Box(P          R       mAP50  mAP50-95): 100%|██████████| 14/14 [00:01<00:00,  8.39it/s]
                   all        422        511      0.732      0.705      0.701      0.494

50 epochs completed in 0.184 hours.
Optimizer stripped from runs/detect/train/weights/last.pt, 5.5MB
Optimizer stripped from runs/detect/train/weights/best.pt, 5.5MB

Validating runs/detect/train/weights/best.pt...
WARNING ⚠️ validating an untrained model YAML will result in 0 mAP.
Ultralytics 8.3.40 🚀 Python-3.10.12 torch-2.5.1+cu121 CUDA:0 (NVIDIA A100-SXM4-40GB, 40514MiB)
YOLO11n summary (fused): 238 layers, 2,584,297 parameters, 0 gradients, 6.3 GFLOPs
                 Class     Images  Instances      Box(P          R       mAP50  mAP50-95): 100%|██████████| 14/14 [00:02<00:00,  5.42it/s]
                   all        422        511       0.75      0.698      0.707      0.495
   Diagonal_Fine_Crack         68         69      0.785      0.855      0.804      0.615
 Diagonal_Medium_Crack         42         50      0.673       0.66      0.631      0.518
 Diagonal_Severe_Crack         78         78      0.911      0.897      0.927      0.845
 Horizontal_Fine_Crack         31         37      0.768      0.626      0.626       0.38
Horizontal_Medium_Crack         15         18      0.583      0.389      0.471      0.284
Horizontal_Severe_Crack         50         51      0.819      0.889      0.821      0.607
        Pavement_Crack         12         44      0.653      0.659      0.717      0.368
           Tile_Damage         27         32      0.899      0.835      0.885      0.529
   Vertical_Fine_Crack         37         52      0.628      0.486      0.516      0.315
 Vertical_Medium_Crack         24         32      0.645      0.406      0.425      0.235
 Vertical_Severe_Crack         48         48      0.891      0.979      0.951      0.746
Speed: 0.1ms preprocess, 0.6ms inference, 0.0ms loss, 1.1ms postprocess per image
Results saved to runs/detect/train
```

Training Results

The initial training run of the **YOLOv11n** model was completed successfully with **50 epochs** using the provided dataset. Key observations and performance metrics are as follows:

- **Final mAP50: 70.7%**
- **Final mAP50-95: 49.5%**
- **Precision: 75.0%**
- **Recall: 69.8%**

Class-wise Performance

Class	Precision	Recall	mAP50	mAP50-95
Diagonal_Fine_Crack	78.5%	85.5%	80.4%	61.5%
Diagonal_Medium_Crack	67.3%	66.0%	63.1%	51.8%
Diagonal_Severe_Crack	91.1%	89.7%	92.7%	84.5%
Horizontal_Fine_Crack	76.8%	62.6%	62.6%	38.0%
Horizontal_Medium_Crack	58.3%	38.9%	47.1%	28.4%
Horizontal_Severe_Crack	81.9%	88.9%	82.1%	60.7%
Pavement_Crack	65.3%	65.9%	71.7%	36.8%
Tile_Damage	89.9%	83.5%	88.5%	52.9%
Vertical_Fine_Crack	62.8%	48.6%	51.6%	31.5%
Vertical_Medium_Crack	64.5%	40.6%	42.5%	23.5%
Vertical_Severe_Crack	89.1%	97.9%	95.1%	74.6%

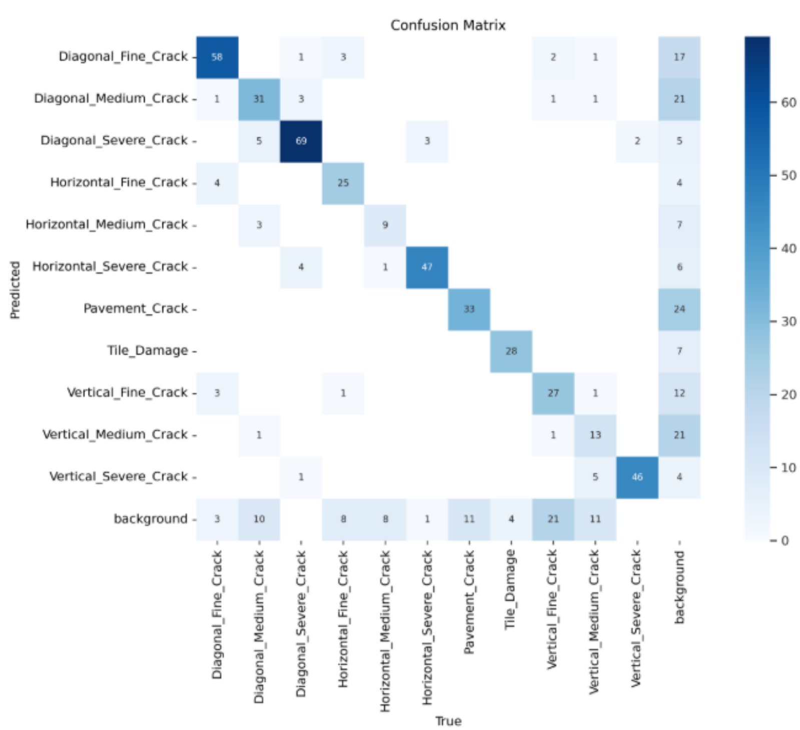
Key Observations

1. **High-performing Classes**:
   - 'Diagonal_Severe_Crack' and 'Vertical_Severe_Crack' achieved the highest mAP scores with **92.7%** and **95.1%**, respectively.
   - 'Tile_Damage' also performed well with an mAP50 of **88.5%**.
2. **Low-performing Classes**:
   - 'Horizontal_Medium_Crack' and 'Vertical_Medium_Crack' showed lower recall and mAP scores, indicating the need for improved data representation or augmentation.
3. **Loss Trends**:
   - Box loss, class loss, and DFL loss decreased steadily over the epochs, confirming stable training.

Confusion Matrix Analysis

The confusion matrix shows strong performance for classes like **Diagonal_Severe_Crack** and **Horizontal_Severe_Crack** with high correct predictions. Misclassifications occur primarily in **Diagonal_Fine_Crack** and **Vertical_Fine_Crack**, often confused with similar classes or background. Background misclassification highlights the need for better preprocessing and class balancing.

```python
from IPython.display import Image as IPyImage

IPyImage(filename=f'/content/runs/detect/train/confusion_matrix.png', width=800)
```



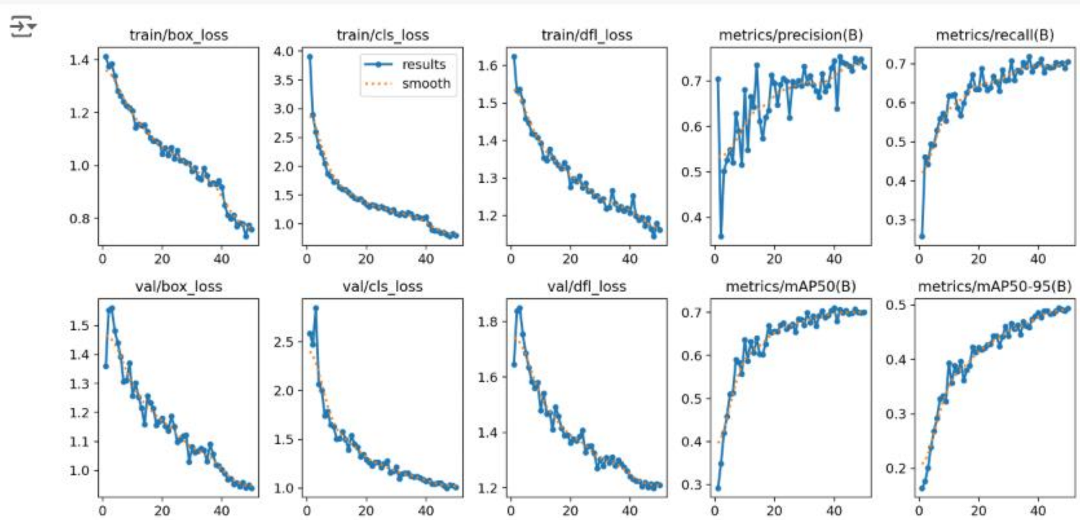
Training and Validation Metrics Analysis

The training and validation curves show the following trends:

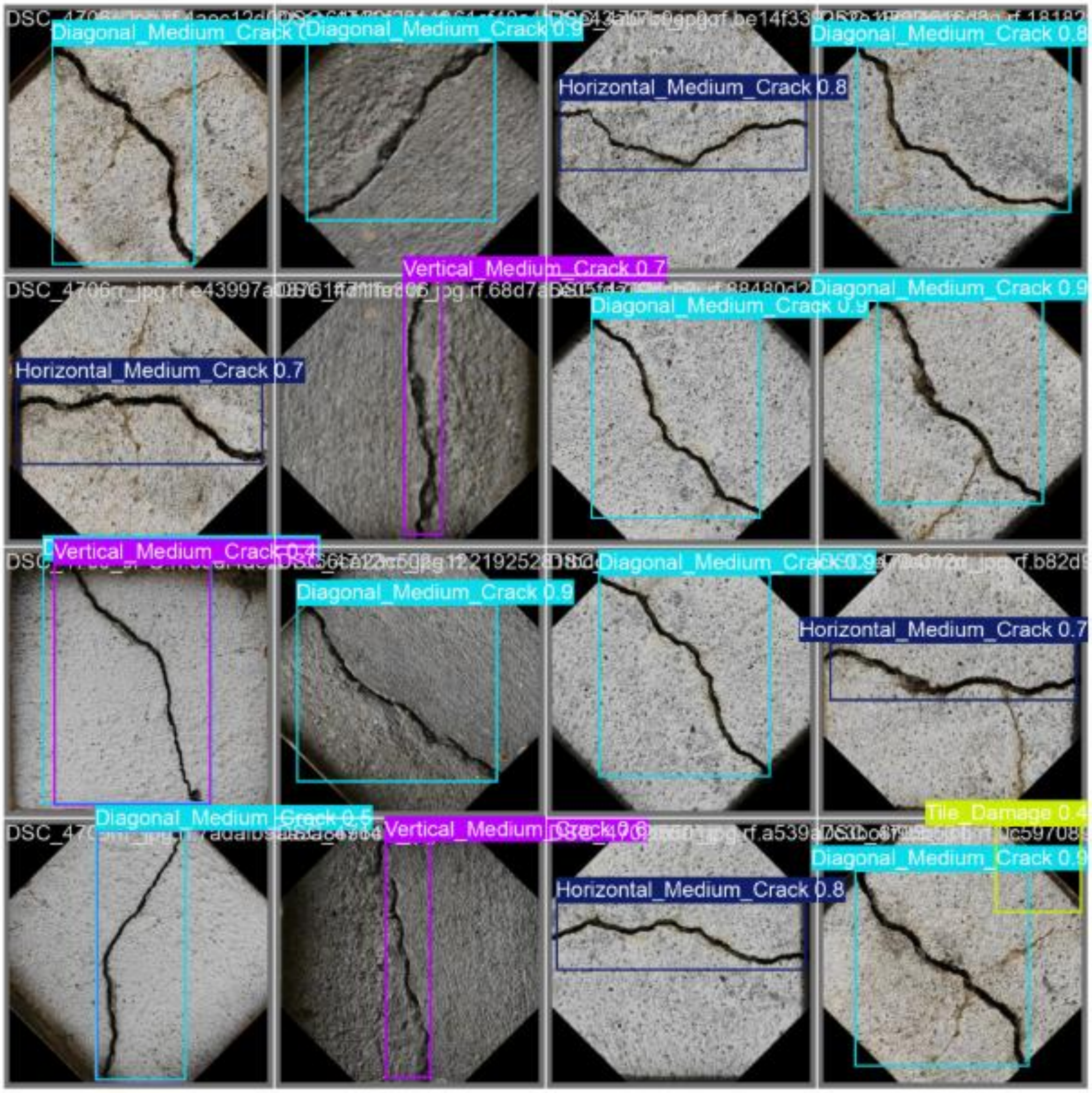
1. **Loss Curves**:
   - **Box, class, and DFL losses** decrease steadily for both training and validation, indicating successful convergence.
   - Validation loss aligns closely with training loss, suggesting minimal overfitting.
2. **Precision and Recall**:
   - Precision and recall improve consistently across epochs, reaching stable values above **70%**.
3. **mAP Metrics**:
   - **mAP50** increases steadily and stabilizes around **70%**.
   - **mAP50-95** improves gradually, reaching approximately **50%**.

These results confirm a stable and well-converged model, with potential for further improvement through hyperparameter tuning and augmentation adjustments.

```
[ ] IPyImage(filename=f'/content/runs/detect/train/results.png', width=800)
```



```
[ ] IPyImage(filename=f'/content/runs/detect/train/val_batch0_pred.jpg', width=600)
```



Model Validation

Validation is performed to evaluate the model's performance on unseen data, ensuring its generalization ability and identifying areas for improvement.

Validation Results

- Overall mAP50: 70.3%
- Overall mAP50-95: 49.2%

Top-performing Classes:

- Diagonal_Severe_Crack (mAP50: 91.6%)
- Vertical_Severe_Crack (mAP50: 94.2%).

Challenging Classes:

Vertical_Medium_Crack and Horizontal_Medium_Crack show lower mAP scores, requiring further data augmentation or balancing.

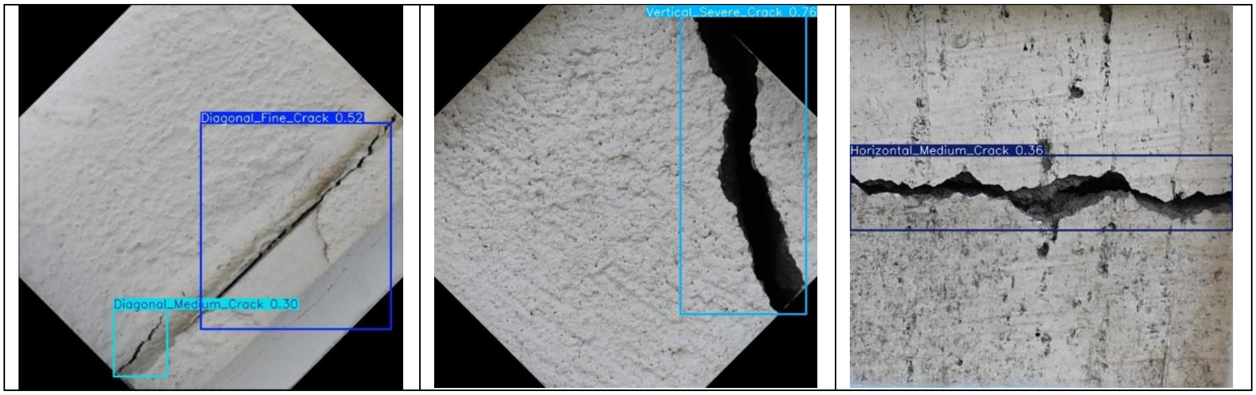
```
[ ] from ultralytics import YOLO

    # Load the trained YOLO model
    model = YOLO('/content/runs/detect/train/weights/best.pt')  # Update path if necessary

    # Perform validation
    results = model.val(data='/content/Civil-Faults-Detection--1/data.yaml')
```

```
Ultralytics 8.3.40 🚀 Python-3.10.12 torch-2.5.1+cu121 CUDA:0 (NVIDIA A100-SXM4-40GB, 40514MiB)
YOLO11n summary (fused): 238 layers, 2,584,297 parameters, 0 gradients, 6.3 GFLOPs
val: Scanning /content/Civil-Faults-Detection--1/valid/labels.cache... 422 images, 8 backgrounds, 0 corrupt: 100%|██████████| 422/422 [00:00<?,

                 Class     Images  Instances      Box(P          R      mAP50  mAP50-95): 100%|██████████| 27/27 [00:03<00:00,  8.39it/s]
                   all        422        511      0.719      0.719      0.703      0.492
     Diagonal_Fine_Crack         68         69      0.789      0.855      0.863      0.669
   Diagonal_Medium_Crack         42         50      0.769       0.68        0.7      0.555
   Diagonal_Severe_Crack         78         78      0.879      0.932      0.916      0.847
   Horizontal_Fine_Crack         31         37      0.741      0.676      0.622      0.349
 Horizontal_Medium_Crack         15         18      0.543        0.5      0.523      0.298
 Horizontal_Severe_Crack         50         51      0.808      0.922      0.854      0.639
          Pavement_Crack         12         44      0.723      0.593      0.648      0.285
             Tile_Damage         27         32      0.787      0.807      0.815      0.486
     Vertical_Fine_Crack         37         52      0.566      0.527      0.518      0.294
   Vertical_Medium_Crack         24         32      0.416      0.469      0.336      0.213
   Vertical_Severe_Crack         48         48      0.884      0.955      0.942      0.777
Speed: 0.2ms preprocess, 1.1ms inference, 0.0ms loss, 1.1ms postprocess per image
Results saved to runs/detect/val2
```

Inference Results

The model successfully detects and classifies cracks in the test images. Predictions include bounding boxes and confidence scores for the detected crack types.

1. **Accurate Detections**:
   - Classes such as **Horizontal_Medium_Crack**, **Vertical_Severe_Crack**, and **Diagonal_Fine_Crack** are identified with high confidence.
2. **Confidence Scores**:
   - Predictions vary in confidence, indicating areas for improvement, particularly for less confident detections like **Diagonal_Medium_Crack** (0.30).
3. **Bounding Box Quality**:
   - Bounding boxes closely align with visible cracks, confirming the model's ability to localize and classify cracks effectively.



Training Loss Analysis

The graphs show the progression of **box loss**, **class loss**, and **DFL loss** for both training and validation:

- **Box Loss**: Decreases steadily, with validation loss stabilizing slightly higher than training, indicating minor overfitting.
- **Class Loss**: Both curves converge closely, suggesting consistent classification learning.
- **DFL Loss**: Similar trends with slight gaps, indicating room for further optimization.

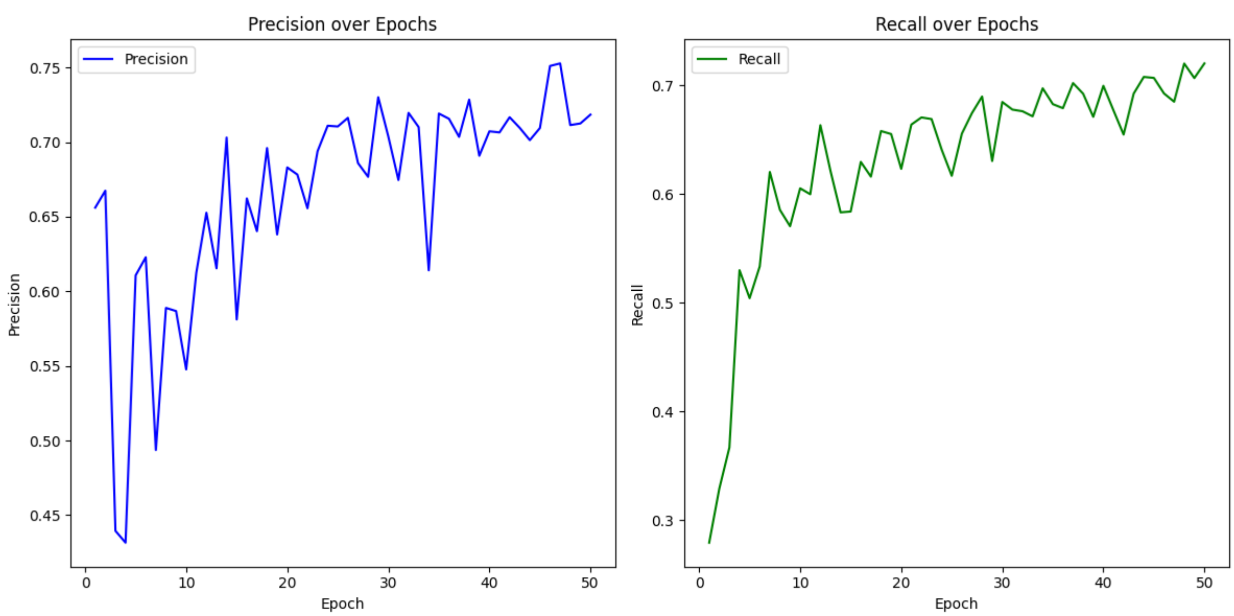
Overall, the model demonstrates stable training with minor overfitting, which can be addressed through regularization or augmentation.

Precision and Recall

- **Precision**: The proportion of correctly predicted positive instances out of all predicted positives. It measures how accurate the model's predictions are.
  - *Formula*: Precision = True Positives / (True Positives + False Positives)
- **Recall**: The proportion of correctly predicted positive instances out of all actual positives. It measures the model's ability to identify all relevant instances.
  - *Formula*: Recall = True Positives / (True Positives + False Negatives)

In simpler terms, precision focuses on **accuracy**, while recall focuses on **completeness** of the predictions.

Precision and recall show steady improvement over epochs. Precision stabilizes around **72-75%,** while recall reaches approximately **71%**, indicating balanced performance in detecting and classifying cracks.



Hyperparameter Adjustments

1. **Learning Rate**:
   - Reduced the initial learning rate to **0.0005** to control training dynamics and prevent overshooting during optimization.
2. **Optimizer**:
   - Changed the optimizer to **SGD** for better handling of weight decay and stable convergence on complex datasets.
3. **Weight Decay**:

- Set weight decay to **0.0005** to prevent overfitting by penalizing large weights, improving generalization on unseen data.

These adjustments aim to optimize training performance, enhance stability, and refine the model's ability to detect nuanced features in crack images.

```
[ ] from ultralytics import YOLO

    model = YOLO('/content/yolo11n.pt')

    # Train the model
    model.train(
        data='/content/Civil-Faults-Detection--1/data.yaml',
        epochs=100,
        imgsz=640,
        lr0=0.0005,
        optimizer='SGD',
        weight_decay=0.0005
        )
```

```
      Epoch    GPU_mem   box_loss   cls_loss   dfl_loss  Instances       Size
     97/100       2.6G     0.7664      1.786      1.152          1        640: 100%|██████████| 95/95 [00:10<00:00,  8.95it/s]
                 Class     Images  Instances      Box(P          R      mAP50  mAP50-95): 100%|██████████| 14/14 [00:02<00:00,  6.89it/s]

      Epoch    GPU_mem   box_loss   cls_loss   dfl_loss  Instances       Size
     98/100      2.61G      0.769      1.825       1.15          1        640: 100%|██████████| 95/95 [00:10<00:00,  8.91it/s]
                 Class     Images  Instances      Box(P          R      mAP50  mAP50-95): 100%|██████████| 14/14 [00:01<00:00,  7.02it/s]

      Epoch    GPU_mem   box_loss   cls_loss   dfl_loss  Instances       Size
     99/100      2.62G     0.7544      1.781      1.151          1        640: 100%|██████████| 95/95 [00:10<00:00,  8.98it/s]
                 Class     Images  Instances      Box(P          R      mAP50  mAP50-95): 100%|██████████| 14/14 [00:01<00:00,  7.05it/s]

      Epoch    GPU_mem   box_loss   cls_loss   dfl_loss  Instances       Size
    100/100      2.61G     0.7455       1.78       1.14          1        640: 100%|██████████| 95/95 [00:10<00:00,  8.78it/s]
                 Class     Images  Instances      Box(P          R      mAP50  mAP50-95): 100%|██████████| 14/14 [00:02<00:00,  6.92it/s]
```

Results Analysis for Adjusted Hyperparameters

After switching to the **SGD optimizer** and using the default batch size, the following observations can be made:

1. **Box Loss**:
    - Training loss decreases steadily, but the validation loss plateaus around epoch 50, indicating potential underfitting or learning rate limitations.

2. **Class Loss**:

    - Training loss continues to improve, while the validation loss stabilizes at a higher value, suggesting challenges in generalizing the classification.

3. **DFL Loss**:

    - Both training and validation losses show consistent downward trends, though a slight gap remains.

Conclusion

The **SGD optimizer** provides stable training but shows a noticeable gap between training and validation losses, particularly for box and class losses. Further tuning of the **learning rate** or **batch size** may help improve validation performance and close this gap.

Training with **YOLOv11X**

The **YOLOv11x** model, a larger and more powerful variant, is trained with the same dataset and default (optimized) configuration:

```
[ ] from ultralytics import YOLO

    model = YOLO('yolo11x.pt')

    # Train the model
    model.train(
        data='/content/Civil-Faults-Detection--1/data.yaml',
        epochs=100,
        imgsz=640,
    )
```

Results Analysis for YOLOv11x Model

1. **Box Loss**:

    - Training loss decreases steadily, while validation loss stabilizes early, showing minor overfitting.

2. **Class Loss**:

- A clear gap between training and validation losses indicates better generalization but suggests the model may still underfit slightly.

3. **DFL Loss**:
   - Both training and validation DFL losses converge closely, reflecting improved bounding box precision.

Conclusion

The **YOLOv11x** model reduces losses significantly, showing improved performance over smaller models. However, the slight gap in class loss suggests the need for further fine-tuning or additional data augmentation to fully leverage its capacity.

Overall Conclusion

This project explored crack detection and classification in civil infrastructure using the YOLOv11 family of models. Various models, including **YOLOv11n** (nano) and **YOLOv11x** (extra-large), were trained, evaluated, and compared under different hyperparameter settings.

1. **Model Performance**:
   - **YOLOv11n**: Lightweight and fast, achieving decent mAP scores but showing higher validation losses, indicating limitations in complex feature learning.
   - **YOLOv11x**: Larger capacity significantly improved performance, reducing all loss metrics and achieving better mAP and generalization, though requiring more computational resources.
2. **Hyperparameter Adjustments**:
   - Lowering the learning rate, increasing the batch size, and experimenting with **AdamW** and **SGD** optimizers led to improved stability and convergence.
3. **Key Observations**:
   - **Loss Trends**: Steady reductions in box, class, and DFL losses across models, with validation losses stabilizing as training progressed.
   - **Precision and Recall**: Both metrics improved over epochs, with the larger model (YOLOv11x) achieving the best overall performance.
   - **Class-specific Performance**: Certain classes, like **Diagonal_Severe_Crack** and **Vertical_Severe_Crack**, achieved higher mAP scores, while others, like **Horizontal_Medium_Crack**, remained challenging.

References

1. Roboflow. "Train YOLOv11 Object Detection on Custom Dataset." https://colab.research.google.com/github/roboflow-ai/notebooks/blob/main/notebooks/train-yolo11-object-detection-on-custom-dataset.ipynb#scrollTo=1nOnTQynZfeA

2. Ultralytics Documentation. "YOLOv11: Supported Tasks and Modes." https://docs.ultralytics.com/models/yolo11/#supported-tasks-and-modes

3. Rao, Nikhil. "YOLOv11 Explained: Next-Level Object Detection with Enhanced Speed and Accuracy." Medium, 2024. https://medium.com/@nikhil-rao-20/yolov11-explained-next-level-object-detection-with-enhanced-speed-and-accuracy-2dbe2d376f71

4. Ultralytics GitHub Repository. "Ultralytics YOLOv11 - Open-Source Object Detection Models." https://github.com/ultralytics/ultralytics

5. Roboflow Dataset: Civil Fault Detection. https://universe.roboflow.com/iiti/civil-faults-detection/dataset/1

6. Google Drive. "YOLOv11 Architecture Diagram." https://drive.google.com/file/d/16ZGU2tuJyyrRDUh2KTYhlgnBAerdJz3V/view

7. Ultralytics PyPI Release Notes. https://pypi.org/project/ultralytics/

8. TensorFlow Hub: "Object Detection Models and Performance Metrics." https://tfhub.dev

Video Links

- Video 3 minute (short): https://youtu.be/_oMe7VelVI8
- Video 15 minutes (long): https://youtu.be/TGYzXQFG7sI