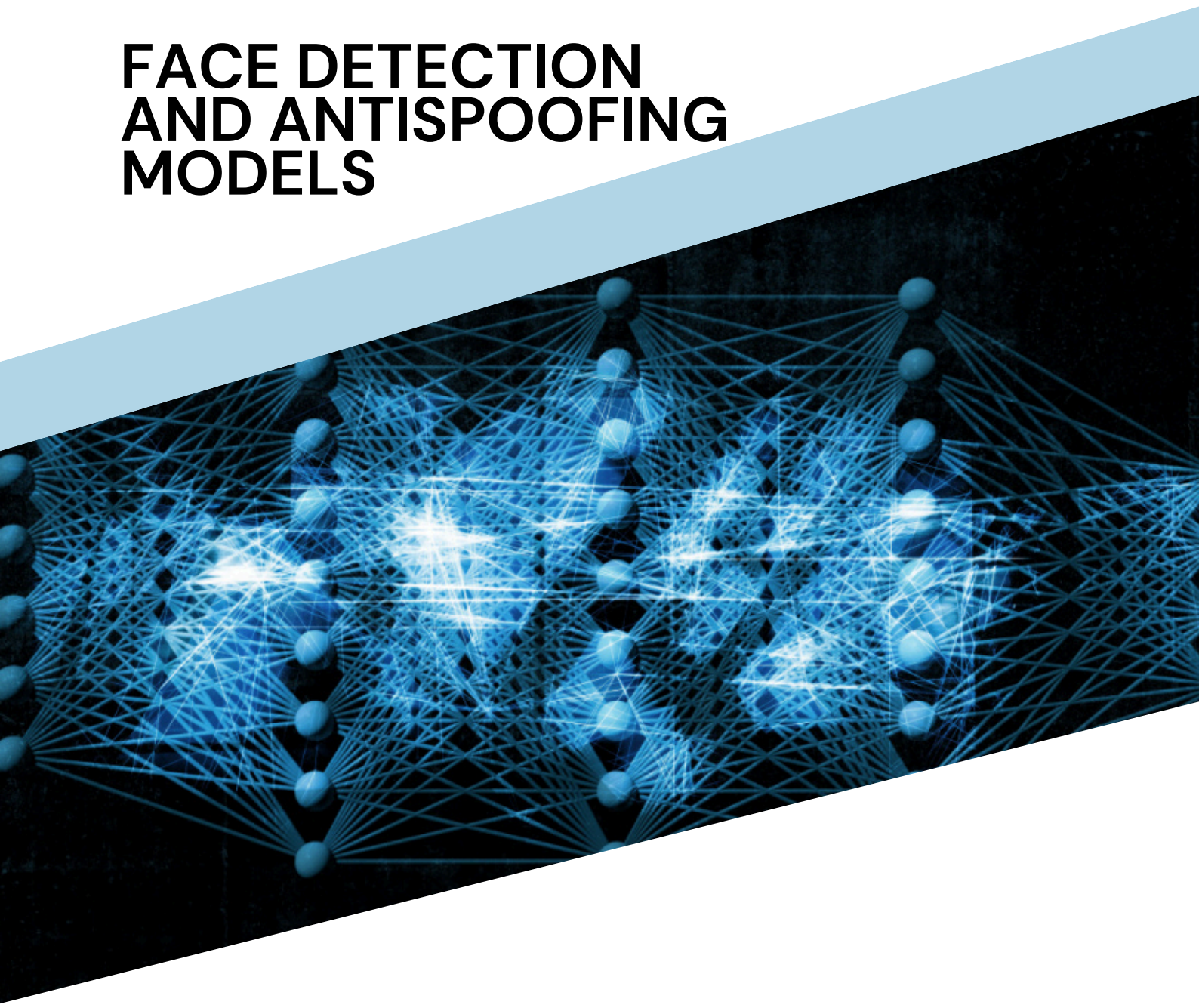
MARCH 2025

FACE DETECTION AND ANTISPOOFING MODELS



PRESENTED BY:

HOUDA ZOUAKI
CAMARA VAMORO
MAMA TOURE RIDIWANE
Hery Jhonny RAZAFINDRAIBE

ENCADRER PAR :

MR. CHAFIK

2025

Table of Contents

Introduction

Facial recognition systems are widely used for authentication but are vulnerable to spoofing attacks using photos, videos, or masks. To enhance security, we propose a robust system that combines face detection with liveness detection.

Our approach involves two models:

1. Face Detection: Using transfert learning with VGG16 to accurately detect faces in images and videos.
2. Liveness Detection: Using YOLO to identify spoofing attempts and ensure the face is real.

By combining these models, we aim to build a secure and reliable facial recognition system.

A-DATA COLLECTION AND LABELLING

1-How do we get our data ?

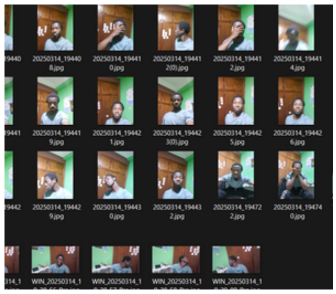
For the data collection part, we chose not to use existing online datasets. Instead, to enhance our learning and understanding of the concept, we decided to build our own dataset from scratch.

Initially, we used our computer's camera along with a Python script to capture approximately 80 images. This hands-on approach allowed us to gain practical experience in data collection while ensuring that our dataset was tailored to our specific needs.

```python
IMAGES_PATH = os.path.join('data','images')
number_images = 30
```

```python
cap = cv2.VideoCapture(1)
for imgnum in range(number_images):
    print('Collecting image {}'.format(imgnum))
    ret, frame = cap.read()
    imgname = os.path.join(IMAGES_PATH,f'{str(uuid.uuid1())}.jpg')
    cv2.imwrite(imgname, frame)
    cv2.imshow('frame', frame)
    time.sleep(0.5)

    if cv2.waitKey(1) & 0xFF == ord('q'):
        break
cap.release()
cv2.destroyAllWindows()
```



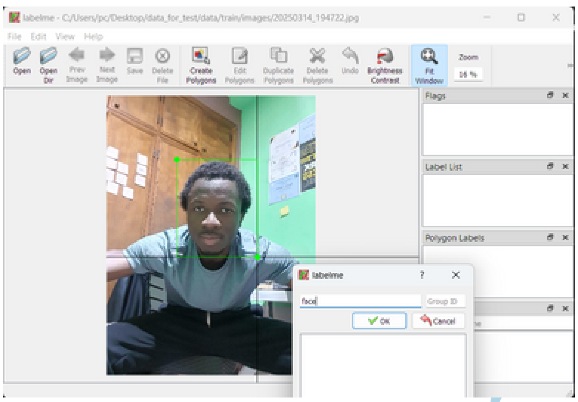
2-Labeled images using labelme ?

LabelMe is an open-source image annotation tool designed for creating high-quality labeled datasets. It allows users to manually draw bounding boxes, polygons, and other shapes around objects in images. One of its key features is that it generates a separate .json file for each annotated image.

These JSON files contain detailed information about the annotations, including:

- Labels: The names assigned to each object (e.g., "face").
- Coordinates: The exact coordinates of the bounding boxes or polygons that enclose the labeled objects.

This structured format makes it easy to parse and use the data during model training, as each JSON file directly maps annotations to the corresponding image. After collecting our images, we used LabelMe to accurately label each face. The generated JSON files provided precise annotations that were essential for training our detection model.



00cb53b298d8431c.json


```json
{
  "version": "5.7.0",
  "flags": {},
  "shapes": [
    {
      "label": "face",
      "points": [
        [
          528.8604651162791,
          347.953488372093O3
        ],
        [
          435.8372093023256,
          207.25581395348837
        ]
      ],
      "group_id": null,
      "description": "",
      "shape_type": "rectangle",
      "flags": {},
      "mask": null
    }
  ],
  "imagePath": "..\\images\\00cb53b298d8431c.jpg",
  "imageData": "/9j/4AAQSkZJRgABAQAAAQABAAD/2wBDAAgGBgcGBQgHBwcJCQgKDBQNDAsLDBk
2wBDAQkJCQwLDBgNDRgyIRwhMjIyMjIyMjIyMjIyMjIyMjIyMjIyMjIyMjIyMjIyMjIyM
```

B-DATA PREPROCESSING

1-Label preprocessing

For the preprocessing step, we started with the labels , specifically the JSON files generated by LabelMe. In these files, the key information we focused on was:

- Label: face / no face
- BBox: Coordinates of the bounding box surrounding the face

We encoded the labels as follows:

- 1 for face
- 0 for no face

For the bounding box coordinates, we normalized them to be between 0 and 1 by dividing:

- xmin and xmax by image.shape[1] (image width)
- ymin and ymax by image.shape[0] (image height)

This normalization is essential because it allows us to use the sigmoid activation function when predicting the coordinates, ensuring the output values are properly scaled.

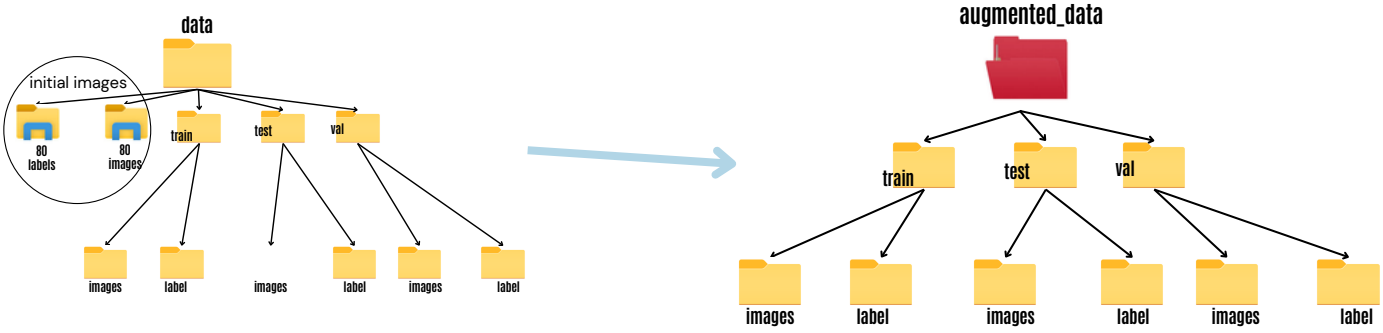
```
[{'label': 'face',
  'points': [[465.3636363636365, 1090.181818181818],
   [1633.5454545454545, 2503.8181818181815]],
  'group_id': None,
  'description': '',
  'shape_type': 'rectangle',
  'flags': {},
  'mask': None}]
```

```
coords
```

```
[0.21076251646903826,
 0.3703063241106719,
 0.7398303689064558,
 0.8504817193675889]
```

2-images preprocessing

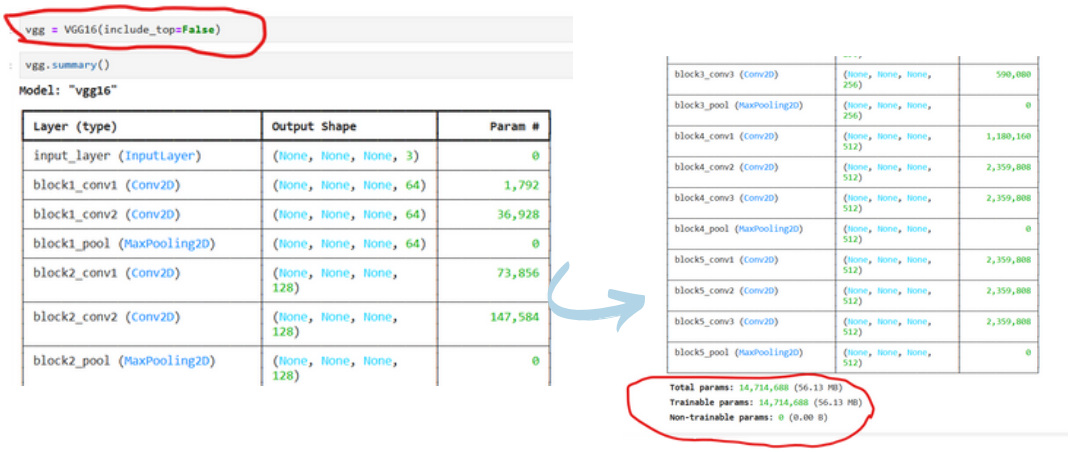
Next, our images and labels, which were initially stored in two separate folders (images and labels), were randomly split into three directories: 70% for training, 15% for testing, and 15% for validation (train, test, and val).
After that, using a Python script and the Albumentations library, we performed data augmentation on the images. Starting with just 80 images, this process resulted in over 4000 images. This augmentation helped us expand the dataset and improve the model's robustness.



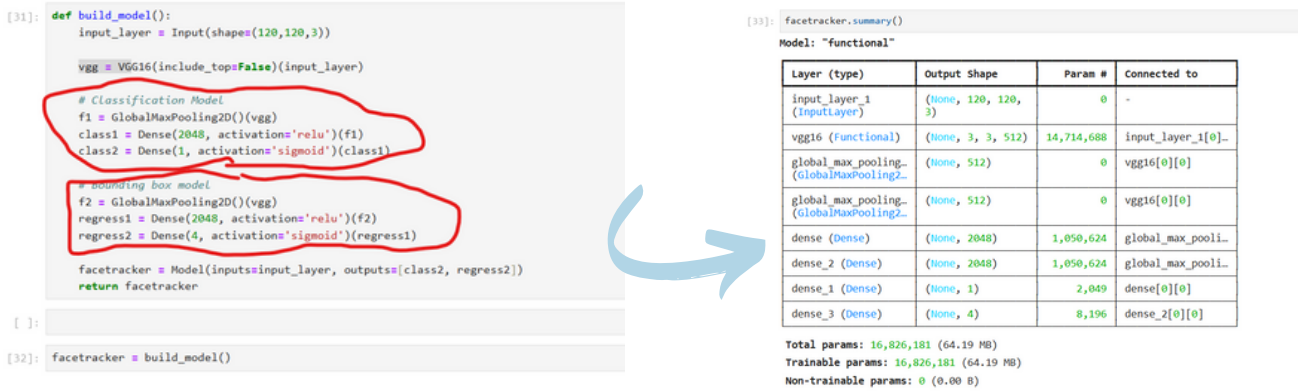
After that, we loaded all the images and their corresponding labels, normalized them, and then placed them into tensors, paired with their respective labels.

C-TRANSFERT LEARNING WITH VGG16 FOR FACE DETECTION AND MODEL EVALUATION

For transfer learning with VGG16, we first loaded the model for feature extraction using the convolutional layers, excluding the dense layers by setting include_top=False. This ensures that we only use the convolutional base of the model. You can also see a summary of the model below:



After that, we built a model based on VGG16 with two outputs: one for classification and the other for regression.



After that, we defined the optimizer and the two loss functions: one for classification and one for regression.

We subclassed the tf.Model class and overridden functions like compile, setting our own parameters and loss functions. Afterward, we created the model and trained it on our dataset.

```
opt = tf.keras.optimizers.Adam(learning_rate=lr_schedule)
```

```
def localization_loss(y_true, yhat):
    delta_coord = tf.reduce_sum(tf.square(y_true[:,:2] - yhat[:,:2]))

    h_true = y_true[:,3] - y_true[:,1]
    w_true = y_true[:,2] - y_true[:,0]

    h_pred = yhat[:,3] - yhat[:,1]
    w_pred = yhat[:,2] - yhat[:,0]

    delta_size = tf.reduce_sum(tf.square(w_true - w_pred) + tf.square(h_true-h_pred))

    return delta_coord + delta_size
```

```
classloss = tf.keras.losses.BinaryCrossentropy()
regressloss = localization_loss
```

```
model = FaceTracker(facetracker)
```

```
model.compile(opt, classloss, regressloss)
```

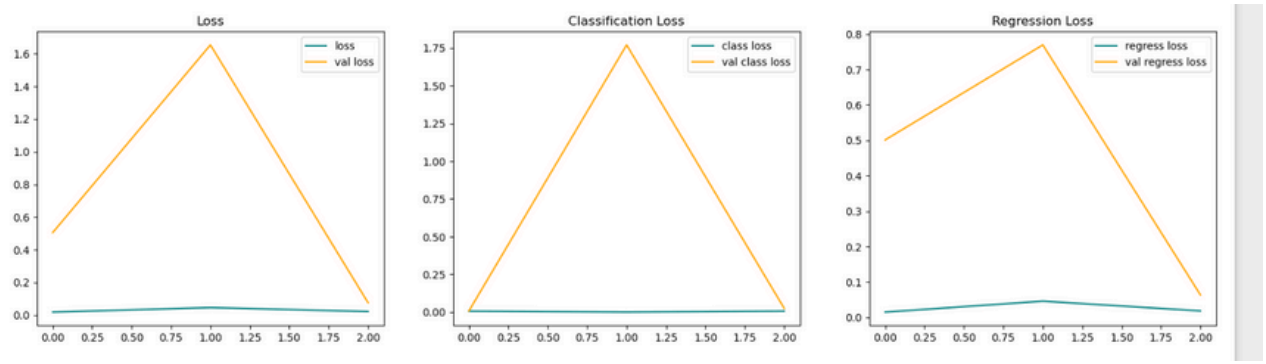
```
logdir='logs'
```

```
tensorboard_callback = tf.keras.callbacks.TensorBoard(log_dir=logdir)
```

```
hist = model.fit(train, epochs=3, validation_data=val, callbacks=[tensorboard_callback])
```

```
Epoch 1/3
296/296 ━━━━━━━━━━━━━━━━━━━━ 475s 2s/step - class_loss: 0.1847 - regress_loss: 0.5442 - total_loss: 0.6365 - val_class
0.6104 - val_total_loss: 0.7603
Epoch 2/3
296/296 ━━━━━━━━━━━━━━━━━━━━ 393s 1s/step - class_loss: 0.1314 - regress_loss: 0.3925 - total_loss: 0.4582 - val_class
0.3650 - val_total_loss: 0.8365
Epoch 3/3
296/296 ━━━━━━━━━━━━━━━━━━━━ 367s 1s/step - class_loss: 0.0902 - regress_loss: 0.2799 - total_loss: 0.3250 - val_class
0.8271 - val_total_loss: 1.1170
```

After training, the sessions took between 1 and 2 hours, depending on the number of epochs, due to the limited performance of our computers. As you can see, the total validation loss, as well as the loss for each model, decreased, which clearly shows that our model is learning effectively.

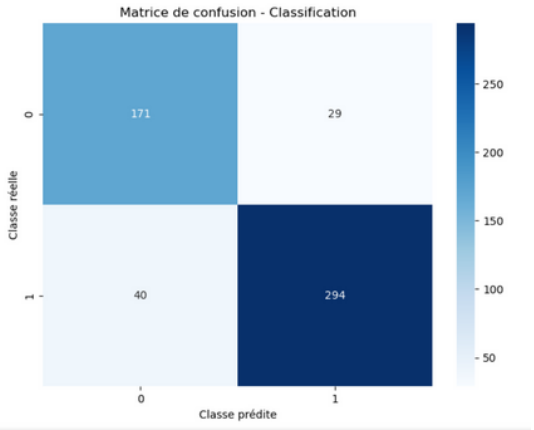


The model performs well, with an accuracy of 87%. It has strong precision and recall for detecting faces (91% and 88%, respectively) and a good balance for "no face" (81% precision, 85% recall). The F1-scores are solid, showing that the model learns well and makes reliable predictions.

```
=== Rapport de classification (Visage ou pas) ===
              precision    recall  f1-score   support

           0       0.81      0.85      0.83       200
           1       0.91      0.88      0.89       334

    accuracy                           0.87       534
   macro avg       0.86      0.87      0.86       534
weighted avg       0.87      0.87      0.87       534
```



The model's regression performance shows good results:

- MAE: 0.1335
- MSE: 0.0614
- RMSE: 0.2478

These values suggest the model's predictions are fairly close to the actual coordinates, but there's still room for improvement.

```
=== Évaluation de la régression (coordonnées) ===
MAE  : 0.1335
MSE  : 0.0614
RMSE : 0.2478
```

D-YOLO MODEL FOR ANTISPOOFING

1-What is YOLO ?

YOLO (You Only Look Once) is a fast and powerful object detection model. Unlike older methods that scan an image multiple times, YOLO looks at the image just once and detects all objects in a single pass. This makes it much faster while still being accurate.

It works by dividing the image into a **grid and predicting** objects within each part. Over the years, different versions (YOLOv1 to YOLOv8) have improved accuracy and efficiency. Today, YOLO is widely used in areas like self-driving cars, security cameras, and robotics because it can detect objects in real time

Key objective: training YOLOv8 to detect if our face is real or fake

2-Dataset Preprocessing

Before training, the dataset needs to be cleaned and formatted to fit YOLO's requirements.
Format of YOLO data : **[classID ,x_center, y_center , width , height]**

2-1 Labling

Each image is assigned a binary label:

0 ------> Fake
1 -----> Real

The dataset is structured in YOLO format, where each image has a corresponding text file containing object class and bounding box coordinates.

```
SplitData > val > labels > 17424104274596.txt
1 0.532031 0.4375 0.282813 0.533333
```

[classID , x_center, y_center , width , height]

Source code found in : **dataCollector.py**

2-2 Normalization

```
# ------  Normalize Values  --------
ih, iw, _ = img.shape
xc, yc = x + w / 2, y + h / 2

xcn, ycn = round(xc / iw, floatingPoint), round(yc / ih, floatingPoint)
wn, hn = round(w / iw, floatingPoint), round(h / ih, floatingPoint)
```

- Image pixel values are normalized to improve model performance.
- Les coordonnées de la boîte englobante sont mises à l'échelle par rapport à la taille de l'image pour une détection appropriée.

Source code found in : **dataCollector.py**

3- Splitting the Dataset

Set	Percentage	Purpose
Training	70%	Model learns from this data
Validation	20%	Used to fine-tune hyperparameters
Test	10%	Evaluates final model performance

```
outputFolderPath = "Dataset/SplitData"
inputFolderPath = "Dataset/All"
splitRatio = {"train": 0.7, "val": 0.2, "test": 0.1}
classes = ["fake","real"]
```

Source code found in : **splitData.py**

4- Training the YOLO Model

To train our model, we use a pretrained **YOLOv8** model (yolov8n.pt) as a starting point and fine-tune it with our dataset.

4-1 Loading YOLOv8

We use Ultralytics' YOLOv8 implementation in Python

```python
from ultralytics import YOLO

model = YOLO('yolov8n.pt')

def main():
    model.train(data='Dataset/SplitData/dataOffline.yaml', epochs=60)


if __name__ == '__main__':
    main()
```

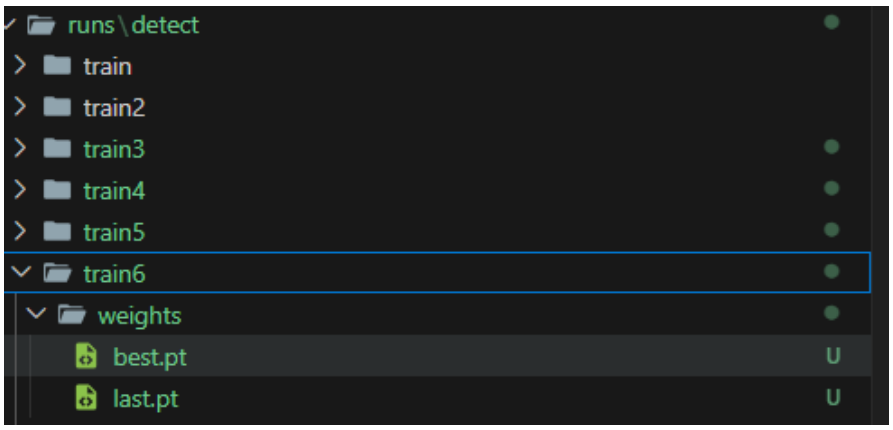
Here, the dataOffline.yaml is the path to reach our Dataset

5- Storing the Trained Model

After training, YOLO saves the best-performing model automatically.

- The final model is stored and used for inference :

```
├── runs/
    ├── detect/
        ├── train/
            ├── weights/
                ├── best.pt  # Best trained model
```



- We can re-name and use **best.pt** for predictions.

6-Model Prediction (Real or Fake Detection)

Once trained, we use the model to classify new images as real or fake.

```python
import math
import time
import cv2
import cvzone
from ultralytics import YOLO

confidence = 0.6

cap = cv2.VideoCapture(0)  # For Webcam
cap.set(3, 640)
cap.set(4, 480)
# cap = cv2.VideoCapture("../Videos/motorbikes.mp4")  # For Videc


model = YOLO("../models/finalversion.pt")

classNames = ["fake", "real"]
```

6-1 Expected Output

- The model provides bounding boxes around detected objects.
- Each object is labeled as real or fake with a confidence score.

Source code found in : **main.py**

7- Results and Performance Analysis

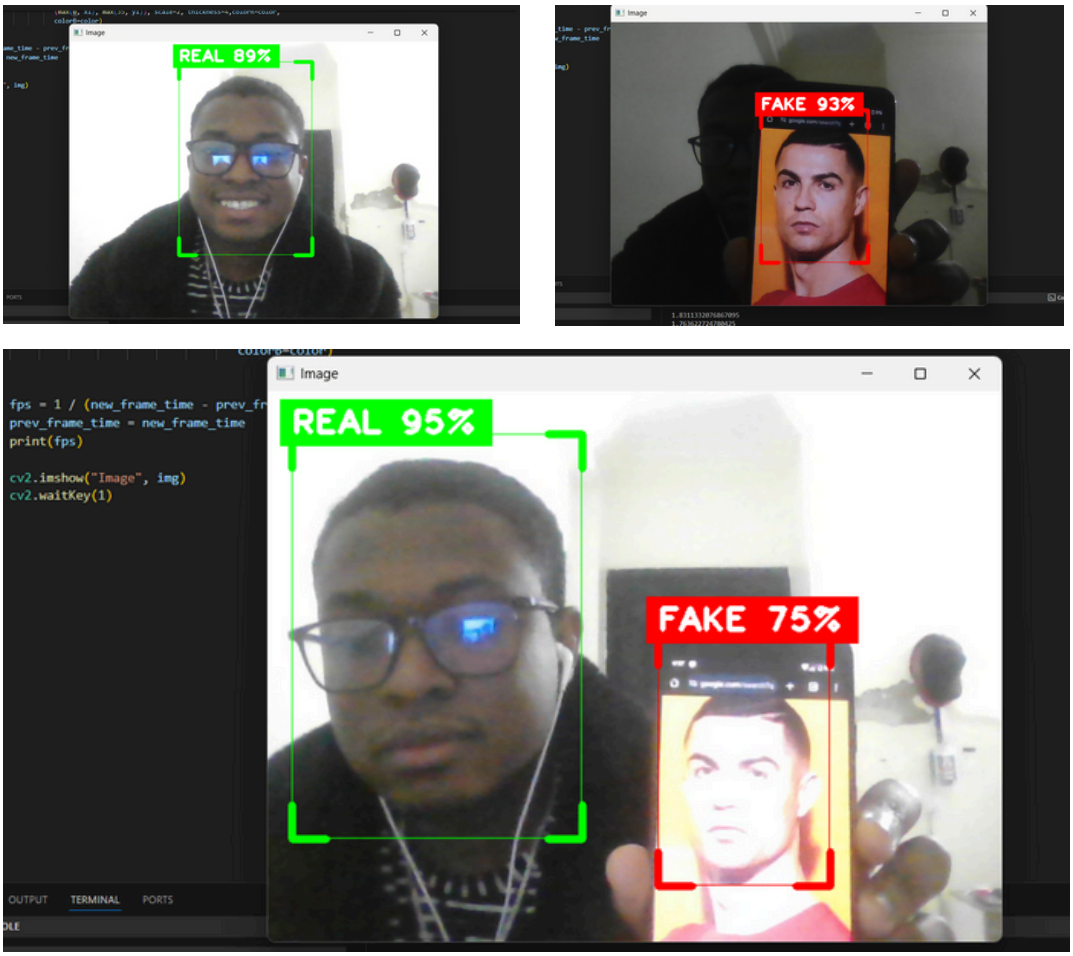
Model Performance

Metrics	Score
Precision	0.995
Recall	0.995
mAP@50	0.995

Path : AntiSpoofing\runs\detect\train6\PR_curve.png

- **The model performs well in distinguishing real vs. fake objects.**
- **Common errors include misclassifications due to poor lighting or occlusion.**

Visualization of Results



Objects are properly classified with high confidence

8-Conclusion

In this project, we successfully combined face detection and liveness detection to build a secure and reliable facial recognition system. By leveraging transfer learning with VGG16 for accurate face detection and using YOLO to detect spoofing attempts, we ensured that the system can not only recognize faces but also distinguish between real and fake ones. The results demonstrate the effectiveness of our approach, paving the way for robust and secure facial authentication solutions.

Key Takeaways

- Our model achieves high accuracy in detecting real vs. fake objects.
- Future improvements could include larger datasets and better augmentation techniques.

For real-world applications, this model could be used in fraud detection, security surveillance, and counterfeit product detection.