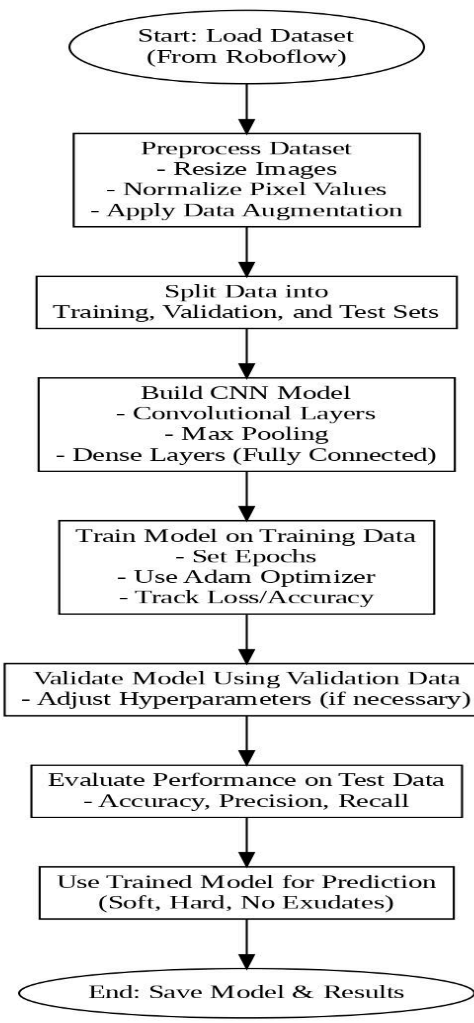
MODEL-2

CNN

Aim: To develop CNN algorithm model to detect the presence of irregularly

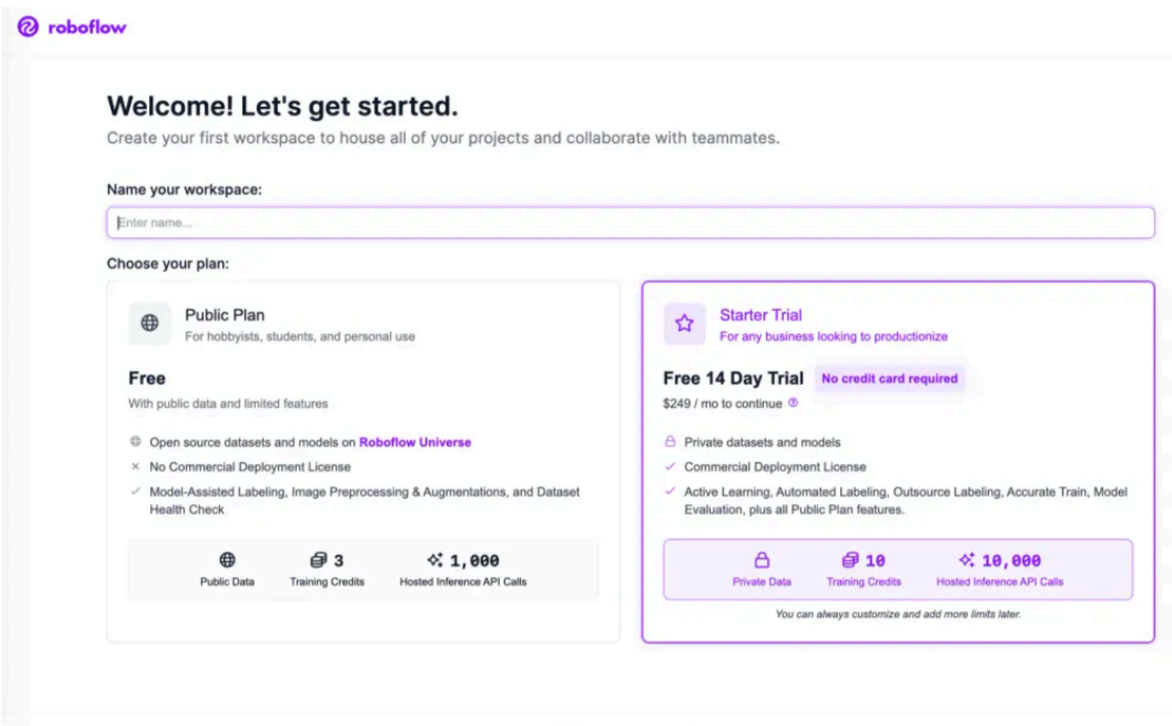
shaped exudates and identify soft , hard and No exudates from retinal fundus image.

Flowchart:



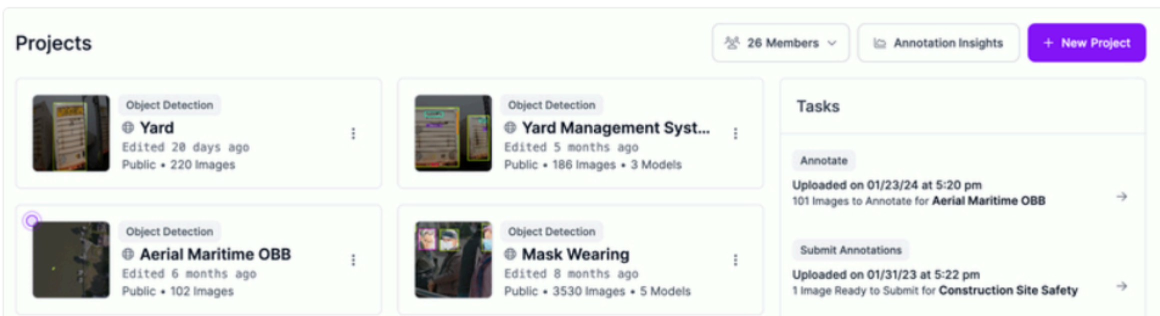
1. Creating an Account on Roboflow

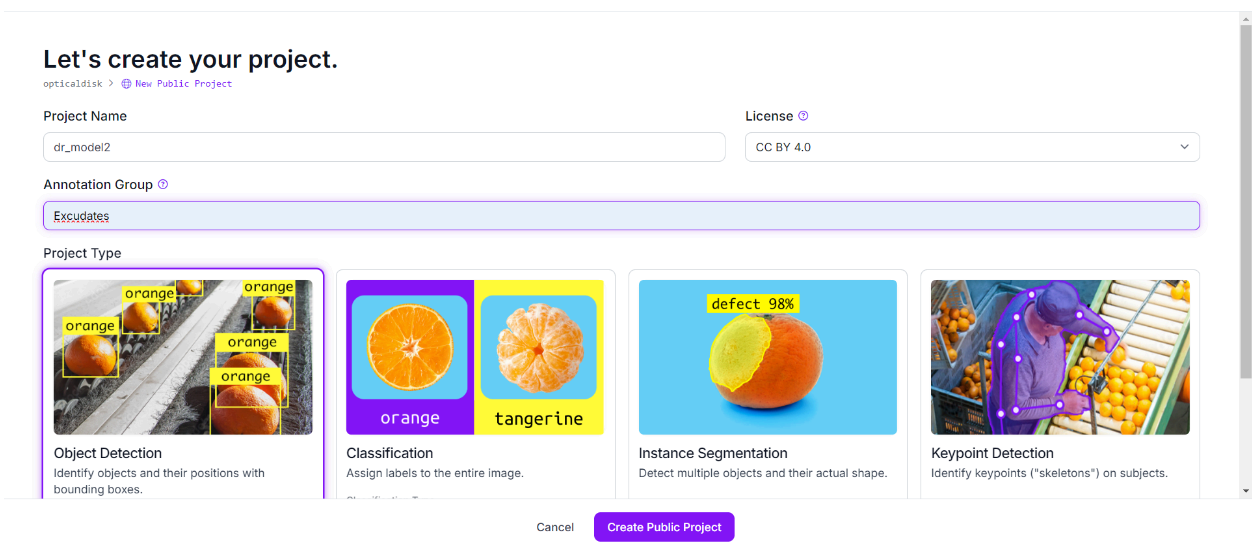
1. **Visit Roboflow Website:**
   - Go to Roboflow.
2. **Sign Up:**
   - Click on the "Sign Up" button in the top right corner.
   - You can sign up using your Google account, GitHub account, or by providing an email and password.
3. **Complete Registration:**
   - Follow the prompts to complete the registration process, including verifying your email if required.



2. Uploading a Dataset

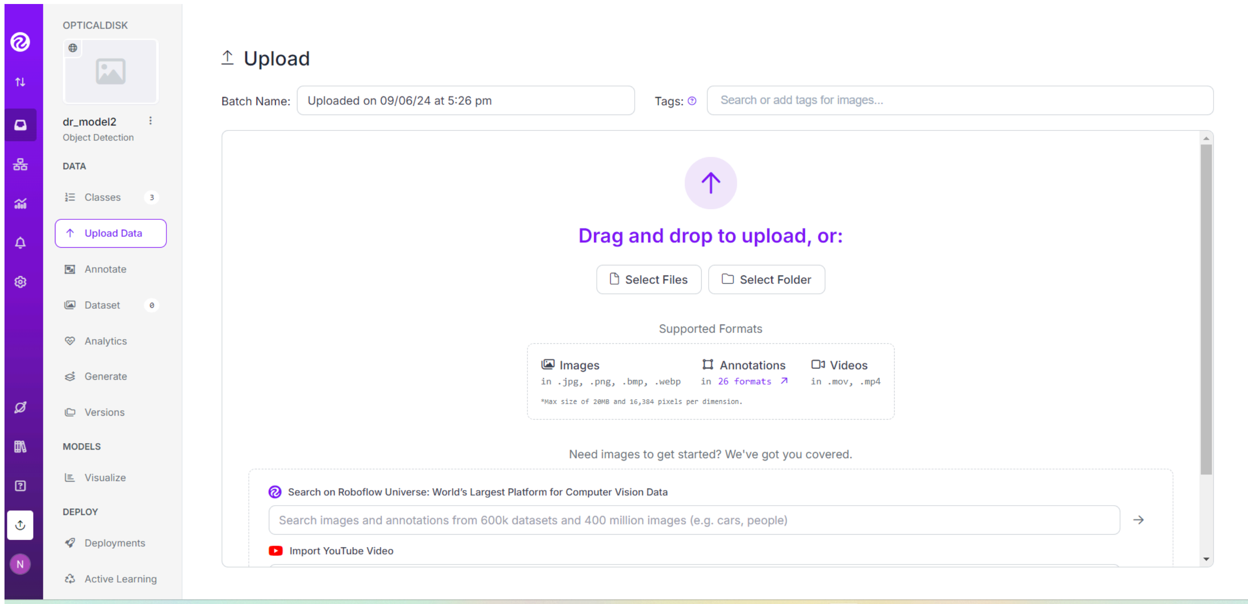
1. **Log In:**
   - Log in to your Roboflow account.
2. **Create a New Project:**
   - Click on "Create New Project" from your dashboard.
   - Fill in the project details like Project Name, Project Type (e.g., Object Detection, Classification, Segmentation), and the License type.
   - Click on "Create Project."





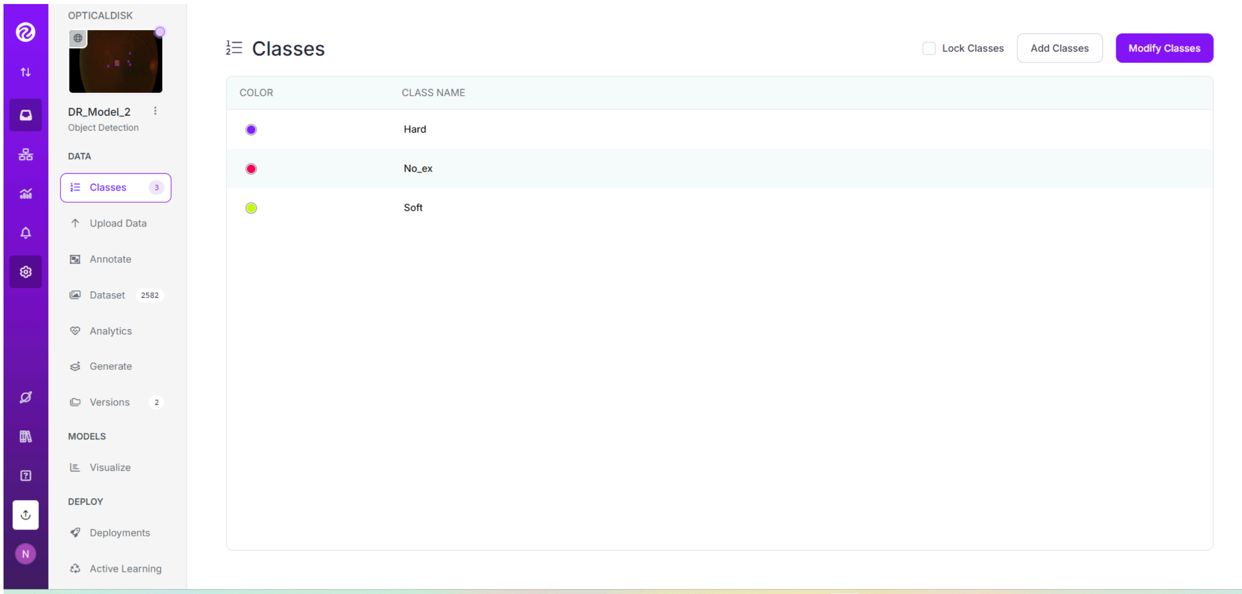
3. **Upload Your Dataset:**
   - Inside the project, click on "Upload Your Images."
   - Drag and drop your images or select files from your computer.
   - Wait for the upload to complete, and click on "Continue" when all files are uploaded.
4. **Configure Dataset Settings:**
   - Set up any pre-processing steps such as resizing or converting to grayscale.

- Click on "Generate" to process the dataset.



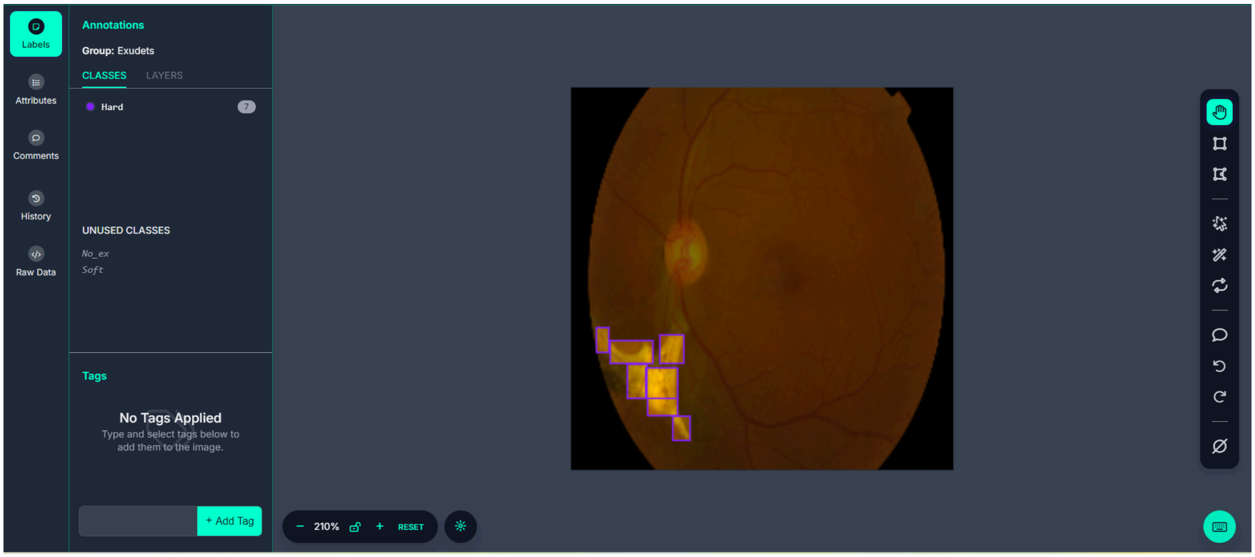
3. Creating and Managing Classes

1. **Access Project Settings:**
    - In your project dashboard, go to the "Dataset" tab.
2. **Add Classes:**
    - Click on "Classes" in the sidebar.
    - To create a new class, click on the "Add Class" button.
    - Enter the name of your class (e.g., "Advanced," "Cotton Wool Spot," "Hard Exudates," etc.).
    - Click "Save" after adding each class.
3. **Edit or Delete Classes:**
    - To edit a class name, click on the pencil icon next to the class name.
    - To delete a class, click on the trash can icon next to the class name.
4. **Organize Classes:**
    - Ensure your classes are correctly organized and named as you need them before proceeding to annotation.



4. Annotating Images

1. **Start Annotating:**
   - Once the dataset is uploaded and classes are defined, click on "Annotate" in your project dashboard.
   - You can either manually annotate images by drawing bounding boxes around objects of interest or use auto-labeling features if available.
2. **Annotation Tools:**
   - **Bounding Box:** Draw rectangles around objects.
   - **Polygon:** Create more precise shapes around objects.
   - **Segmentation:** For detailed object outlines.
   - Assign each annotation to one of the classes you created earlier.
3. **Save Annotations:**
   - After annotating each image, click "Save" to store the annotations.
   - Continue annotating until all images are labeled.

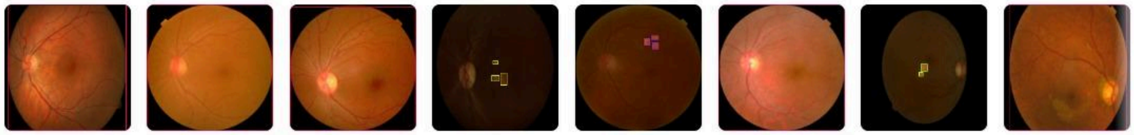


5. Training a Model

1. **Prepare Your Dataset:**
   - Once all annotations are complete, click on the "Versions" tab in your project.
   - Click "Create New Version" and ensure all images and annotations are included.
2. **Select a Model:**
   - Click on "Train a Model" from the project dashboard.
   - Roboflow will guide you through setting up a model training job.
   - Select the appropriate model architecture (e.g., YOLOv5, Faster R-CNN) based on your project needs.
3. **Configure Training:**
   - Set training parameters like epochs, learning rate, and batch size.
   - Select the split for training, validation, and testing (usually Roboflow defaults to an 80-20 split).
4. **Start Training:**
   - Click "Start Training" to begin.
   - Roboflow will automatically handle the training process and provide real-time updates on progress.
5. **Monitor Training:**
   - Check the metrics such as loss, accuracy, and mAP (mean Average Precision) during training.
   - If the results are not satisfactory, you may adjust the parameters and retrain the model.

5008 Total Images

View All Images →



Dataset Split

TRAIN SET 85%	VALID SET 10%	TEST SET 5%
4233 Images	516 Images	259 Images

Preprocessing

Auto-Orient: Applied

Resize: Stretch to 640×640

4 **Augmentation**

What can augmentation do?

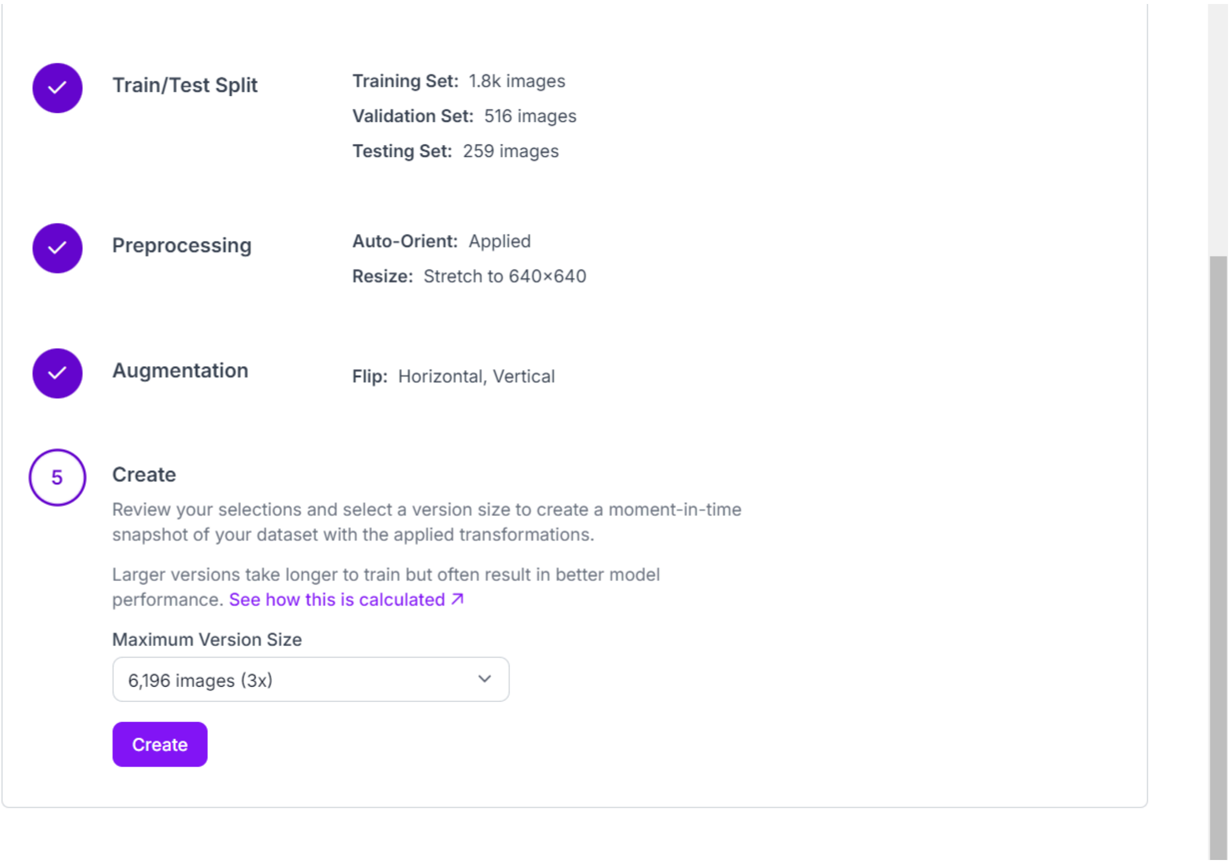
Create new training examples for your model to learn from by generating augmented versions of each image in your training set.

Flip
Horizontal, Vertical
Edit ×

Add Augmentation Step

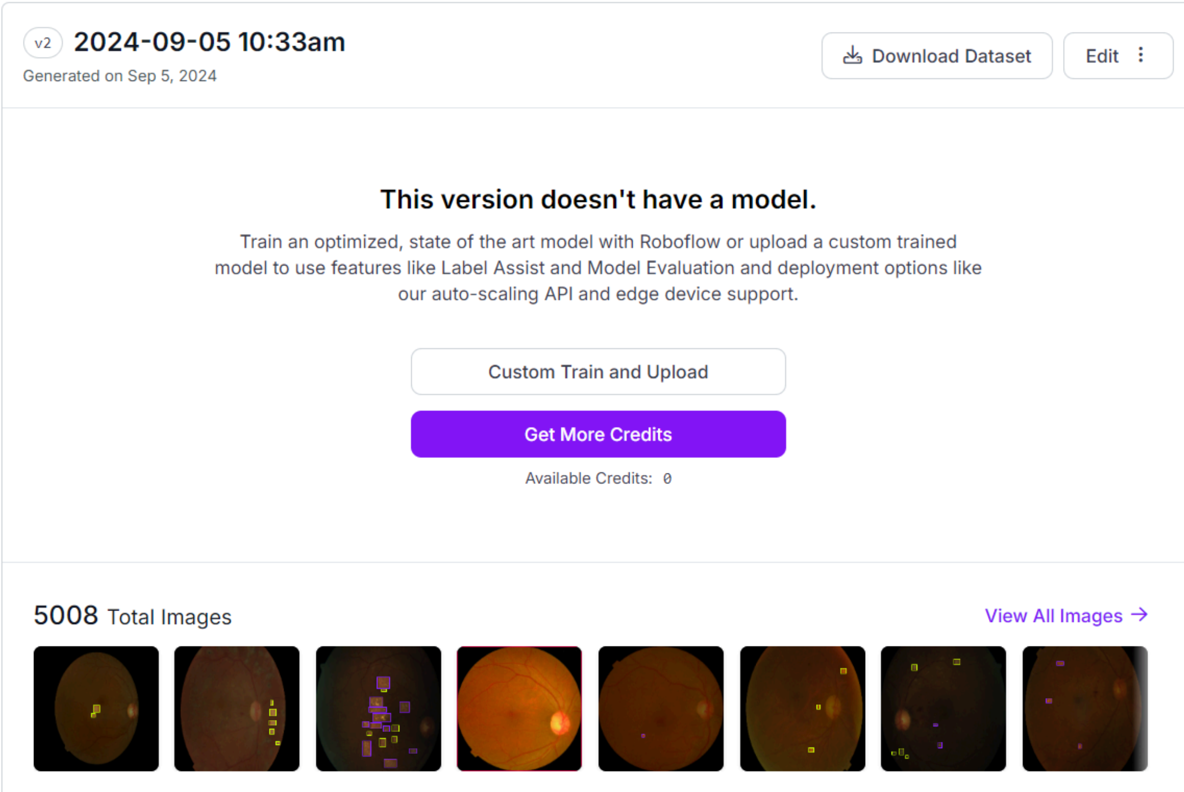
Continue

5 Create

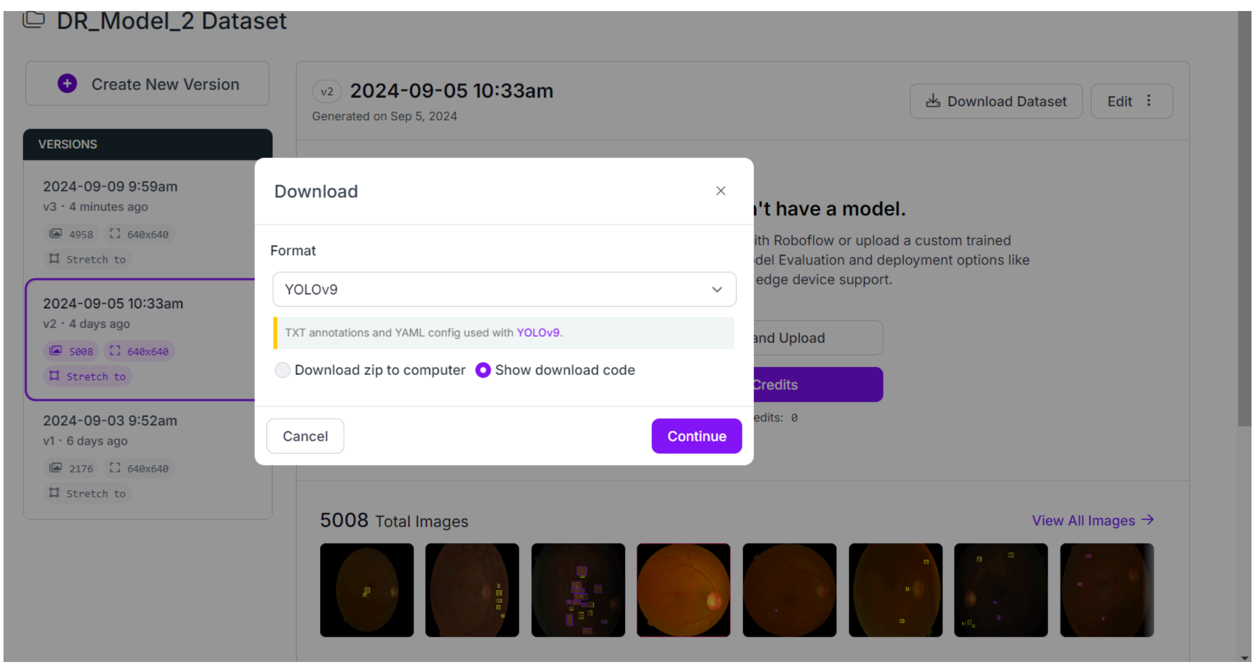


6. Deploying and Using the Model

1. **Deploy the Model:**
   - Once training is complete, you can deploy the model directly from Roboflow.
   - Click on "Deploy" in your project dashboard to get the model's API endpoint and download options.
2. **Using the Model:**
   - Roboflow provides multiple deployment options including Web API, mobile SDKs, or exporting the model weights for local inference.
3. **Test the Model:**
   - Use the test dataset or real-world images to test your model's performance.



Click to download dataset:

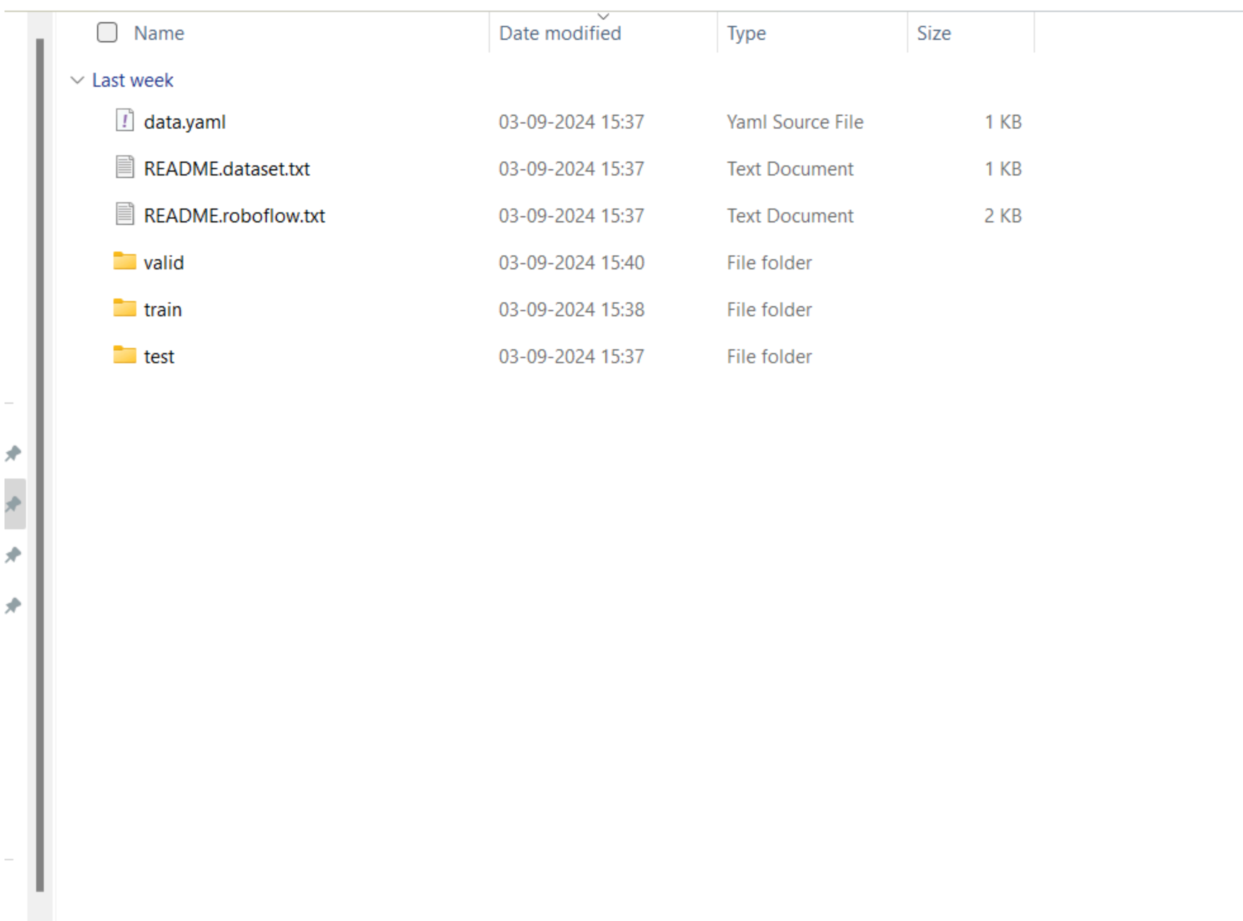


Select format and after select structure that is zip folder or code .

Additional Tips

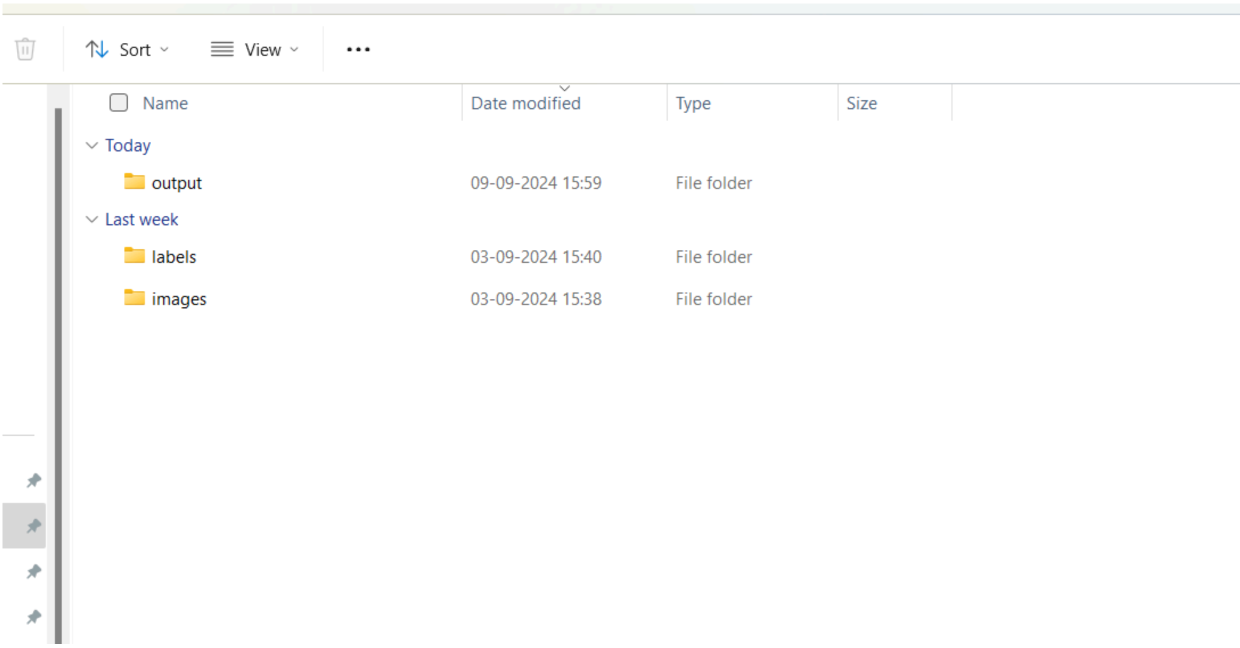
- **Project Settings:** Regularly update the dataset and retrain the model with new data to keep it relevant.
- **Collaborate:** Invite team members to collaborate on the project for annotating and training.
- **API Integration:** Roboflow provides easy integration with various platforms and frameworks for seamless deployment.

Step 2: Extraction code:



code processes images and their corresponding YOLO-formatted label files to extract and resize objects based on the bounding boxes specified in the labels. Here's a detailed step-by-step breakdown of what the code does:

1. Setting Up File Paths
The script starts by defining paths for images, labels, and output folders:
Python



In train, valid and test folder create new output folder to save output classes images

Copy code:

```
images_folder = "C://Users//Niharika//Downloads//DR_Model_2_1//test//images//"
labels_folder = "C://Users//Niharika//Downloads//DR_Model_2_1//test//labels//"
output_folder = "C://Users//Niharika//Downloads//DR_Model_2_1//test//output_test//"
```

- These paths indicate where the images and labels are stored and where the processed images will be saved.

2. Defining Constants
The code specifies a fixed size (224, 224) for resizing the output images:
python
Copy code

```
fixed_size = (224, 224)
```

-

A dictionary class_names maps class IDs to their respective names:
python

Copy code

```
class_names = {
    0: "Hard",
    1: "No_ex",
    2: "Soft"
}
```

-

3. Creating the Output Folder
Before processing, it checks if the output folder exists and creates it if not:
python
Copy code

```
if not os.path.exists(output_folder):
    os.makedirs(output_folder)
```

4. Function to Convert YOLO Format to Bounding Box Coordinates
The function convert_yolo_to_bbox is used to convert YOLO annotations (normalized coordinates) into actual bounding box coordinates for cropping:
python
Copy code:

```
def convert_yolo_to_bbox(img_width, img_height, x_center, y_center, width, height):
    x_center *= img_width
    y_center *= img_height
    width *= img_width
    height *= img_height

    x_min = int(x_center - width / 2)
    y_min = int(y_center - height / 2)
    x_max = int(x_center + width / 2)
    y_max = int(y_center + height / 2)

    return x_min, y_min, x_max, y_max
```

5. Processing Each Image
The code iterates over each file in the images_folder:
python
Copy code

```
for image_filename in os.listdir(images_folder):
    # Construct paths for images and corresponding labels
    image_path = os.path.join(images_folder, image_filename)
```

```
    label_filename = image_filename.replace('.png', '.txt').replace('.jpg', '.txt')
    label_path = os.path.join(labels_folder, label_filename)
```

- It constructs paths for the image and its corresponding label file.

6. Checking for the Label File
For each image, the script checks if a corresponding label file exists:
python
Copy code

```
if not os.path.isfile(label_path):
    print(f"Label file not found for image: {image_filename}")
    continue
```

- If the label file is missing, the image is skipped.

7. Reading the Image and Label File
The script opens the image using PIL:
python
Copy code

```
with Image.open(image_path) as img:
    img_width, img_height = img.size
```

It reads the label file line by line, expecting each line to contain class IDs and bounding box information in YOLO format:
python
Copy code:

```
with open(label_path, 'r', encoding='utf-8') as label_file:
    for line in label_file:
        parts = line.strip().split()
        class_id = int(parts[0])
        x_center, y_center, width, height = map(float, parts[1:])
```

-

8. Converting YOLO Format and Cropping the Image

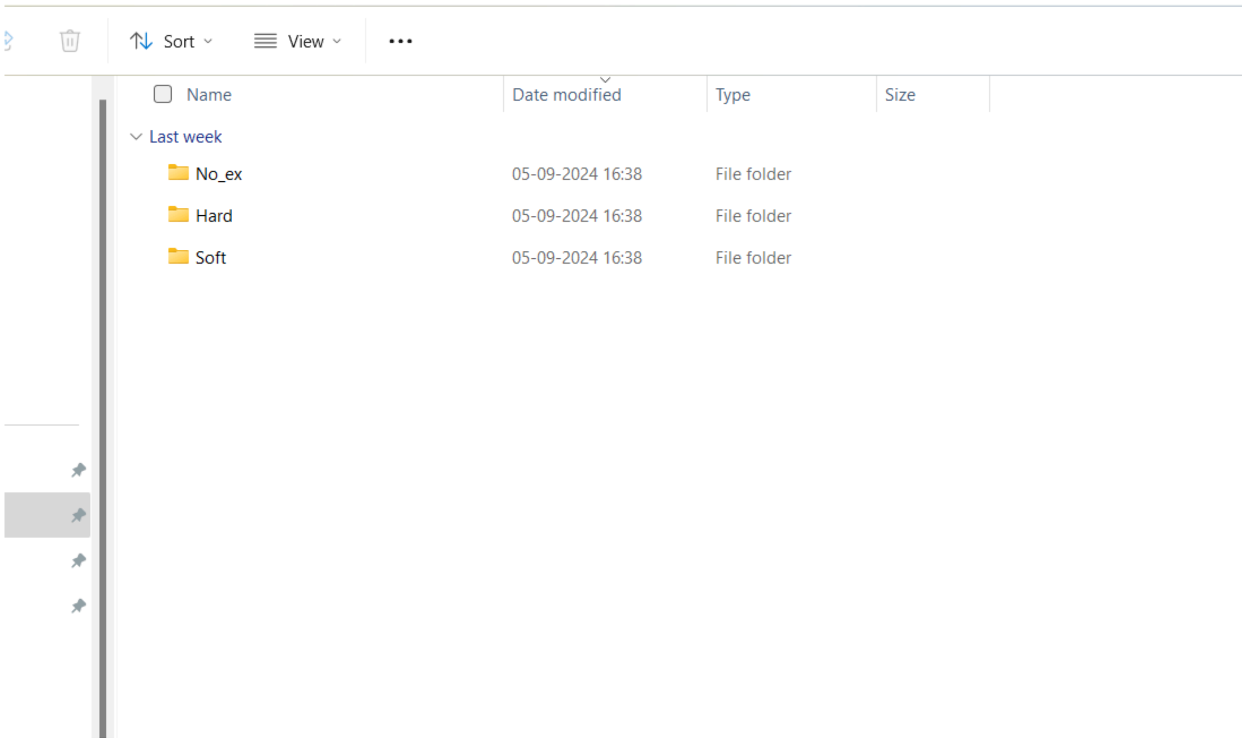
- Each line in the label file is processed to extract bounding box coordinates using the convert_yolo_to_bbox function.
- The bounding box coordinates are adjusted to ensure they stay within the image boundaries.

The image is cropped using these coordinates:
python
Copy code
```
cropped_img = img.crop((x_min, y_min, x_max, y_max))
```

9. Resizing and Saving the Cropped Image
The cropped image is resized to the fixed size (224, 224) using Lanczos resampling for high quality:
python
Copy code:
```
resized_img = cropped_img.resize(fixed_size, Image.Resampling.LANCZOS)
```

The resized image is saved into a folder corresponding to its class name:
python
Copy code:
```
class_folder = os.path.join(output_folder, class_name)
if not os.path.exists(class_folder):
    os.makedirs(class_folder)
cropped_image_filename = f"{os.path.splitext(image_filename)[0]}_{x_min}_{y_min}.jpg"
cropped_image_path = os.path.join(class_folder, cropped_image_filename)
resized_img.save(cropped_image_path)
print(f"Saved resized object to {cropped_image_path}")
```

10. Handling Errors
If there is an issue with reading the label file due to encoding errors, the script catches and prints the error:
python
Copy code
```
except UnicodeDecodeError as e:
    print(f"Error reading file {label_path}: {e}")
```



-

Summary

- The code essentially loops through images, looks for corresponding YOLO label files, converts the YOLO format bounding boxes into image-specific coordinates, crops and resizes the objects based on these boxes, and then saves the processed images into class-specific folders.
- It ensures that the objects are resized uniformly to (224, 224), making the data ready for further processing, such as training a CNN model.

Whole code:

```
import os

from PIL import Image



# Define paths

#images_folder = "E://model2//images//"

#labels_folder = "E://model2//labels//"

#output_folder = "E:\model2\output"
```

```
#images_folder = "C://Users//Niharika//Downloads//DR_Model_2_1//valid//images//"
#labels_folder = "C://Users//Niharika//Downloads//DR_Model_2_1//valid//labels//"
#output_folder = "C://Users//Niharika//Downloads//DR_Model_2_1//valid//output//"

#images_folder = "C://Users//Niharika//Downloads//DR_Model_2_1//train//images//"
#labels_folder = "C://Users//Niharika//Downloads//DR_Model_2_1//train//labels//"
#output_folder = "C://Users//Niharika//Downloads//DR_Model_2_1//train//output//"

labels_folder = "C://Users//Niharika//Downloads//DR_Model_2_1//test//labels//"
images_folder="C://Users//Niharika//Downloads//DR_Model_2_1//test//images//"
output_folder = "C://Users//Niharika//Downloads//DR_Model_2_1//test//output_test//"


# Define the fixed size for output images
fixed_size = (224, 224)

# Define class names based on IDs
class_names = {
    0: "Hard",
    1: "No_ex",
    2: "Soft"
}
```

```
# Create output folder if it doesn't exist
if not os.path.exists(output_folder):
    os.makedirs(output_folder)


def convert_yolo_to_bbox(img_width, img_height, x_center, y_center, width, height):
    # Convert YOLO format to bounding box coordinates
    x_center *= img_width
    y_center *= img_height
    width *= img_width
    height *= img_height

    x_min = int(x_center - width / 2)
    y_min = int(y_center - height / 2)
    x_max = int(x_center + width / 2)
    y_max = int(y_center + height / 2)

    return x_min, y_min, x_max, y_max

# Iterate over files in the images folder
for image_filename in os.listdir(images_folder):
    # Construct the path to the image file
    image_path = os.path.join(images_folder, image_filename)

    # Construct the corresponding label file path (assuming label is a .txt file)
```

```
    label_filename = image_filename.replace('.png', '.txt').replace('.jpg', '.txt')  # Adjust extensions as needed

    label_path = os.path.join(labels_folder, label_filename)


    # Debug information

    print(f"Processing image: {image_path}")

    print(f"Looking for label: {label_path}")


    # Check if the label file exists

    if not os.path.isfile(label_path):

        print(f"Label file not found for image: {image_filename}")

        continue  # Skip to the next image if the label file is missing


    # Open the image

    with Image.open(image_path) as img:

        img_width, img_height = img.size


        # Read the label file

        try:

            with open(label_path, 'r', encoding='utf-8') as label_file:

                for line in label_file:

                    parts = line.strip().split()

                    class_id = int(parts[0])

                    x_center, y_center, width, height = map(float, parts[1:])
```

```
# Convert YOLO format to bounding box coordinates
x_min, y_min, x_max, y_max = convert_yolo_to_bbox(img_width, img_height, x_center,
y_center, width, height)

# Ensure coordinates are within the image bounds
x_min = max(0, x_min)
y_min = max(0, y_min)
x_max = min(img_width, x_max)
y_max = min(img_height, y_max)

# Crop the image
cropped_img = img.crop((x_min, y_min, x_max, y_max))

# Resize the cropped image to the fixed size
resized_img = cropped_img.resize(fixed_size, Image.Resampling.LANCZOS)

# Map class_id to class name
class_name = class_names.get(class_id, "Unknown")

# Create a folder for the class if it doesn't exist
class_folder = os.path.join(output_folder, class_name)
if not os.path.exists(class_folder):
    os.makedirs(class_folder)

# Save the resized image
```

cropped_image_filename = f"{os.path.splitext(image_filename)[0]}_{x_min}_{y_min}.jpg" # Adjust naming as needed

cropped_image_path = os.path.join(class_folder, cropped_image_filename)

resized_img.save(cropped_image_path)

print(f"Saved resized object to {cropped_image_path}")

except UnicodeDecodeError as e:

print(f"Error reading file {label_path}: {e}")

File Edit Format Run Options Window Help

```
import os
from PIL import Image

# Define paths
#images_folder = "E://model2//images//"
#labels_folder = "E://model2//labels//"
#output_folder = "E:\model2\output"

#images_folder = "C://Users//Niharika//Downloads//DR_Model_2_1//valid//images//"
#labels_folder = "C://Users//Niharika//Downloads//DR_Model_2_1//valid//labels//"
#output_folder = "C://Users//Niharika//Downloads//DR_Model_2_1//valid//output//"

#images_folder = "C://Users//Niharika//Downloads//DR_Model_2_1//train//images//"
#labels_folder = "C://Users//Niharika//Downloads//DR_Model_2_1//train//labels//"
#output_folder = "C://Users//Niharika//Downloads//DR_Model_2_1//train//output//"

labels_folder = "C://Users//Niharika//Downloads//DR_Model_2_1//test//labels//"
images_folder="C://Users//Niharika//Downloads//DR_Model_2_1//test//images//"
output_folder = "C://Users//Niharika//Downloads//DR_Model_2_1//test//output_test//"


# Define the fixed size for output images
fixed_size = (224, 224)

# Define class names based on IDs
class_names = {
    0: "Hard",
    1: "No_ex",
    2: "Soft"
}

# Create output folder if it doesn't exist
if not os.path.exists(output_folder):
    os.makedirs(output_folder)

def convert_yolo_to_bbox(img_width, img_height, x_center, y_center, width, height):
    # Convert YOLO format to bounding box coordinates
    x_center *= img_width
    y_center *= img_height
    width *= img_width
    height *= img_height

    x_min = int(x_center - width / 2)
    y_min = int(y_center - height / 2)
    x_max = int(x_center + width / 2)
    y_max = int(y_center + height / 2)
```

```python
    return x_min, y_min, x_max, y_max

# Iterate over files in the images folder
for image_filename in os.listdir(images_folder):
    # Construct the path to the image file
    image_path = os.path.join(images_folder, image_filename)

    # Construct the corresponding label file path (assuming label is a .txt file)
    label_filename = image_filename.replace('.png', '.txt').replace('.jpg', '.txt')  # Adjust extensions as needed
    label_path = os.path.join(labels_folder, label_filename)

    # Debug information
    print(f"Processing image: {image_path}")
    print(f"Looking for label: {label_path}")

    # Check if the label file exists
    if not os.path.isfile(label_path):
        print(f"Label file not found for image: {image_filename}")
        continue  # Skip to the next image if the label file is missing

    # Open the image
    with Image.open(image_path) as img:
        img_width, img_height = img.size

        # Read the label file
        try:
            with open(label_path, 'r', encoding='utf-8') as label_file:
                for line in label_file:
                    parts = line.strip().split()
                    class_id = int(parts[0])
                    x_center, y_center, width, height = map(float, parts[1:])

                    # Convert YOLO format to bounding box coordinates
                    x_min, y_min, x_max, y_max = convert_yolo_to_bbox(img_width, img_height, x_center, y_center, width, height)

                    # Ensure coordinates are within the image bounds
                    x_min = max(0, x_min)
                    y_min = max(0, y_min)
                    x_max = min(img_width, x_max)
                    y_max = min(img_height, y_max)

                    # Crop the image
                    cropped_img = img.crop((x_min, y_min, x_max, y_max))

                    # Resize the cropped image to the fixed size
                    resized_img = cropped_img.resize(fixed_size, Image.Resampling.LANCZOS)
```


```python
                    x_min = max(0, x_min)
                    y_min = max(0, y_min)
                    x_max = min(img_width, x_max)
                    y_max = min(img_height, y_max)

                    # Crop the image
                    cropped_img = img.crop((x_min, y_min, x_max, y_max))

                    # Resize the cropped image to the fixed size
                    resized_img = cropped_img.resize(fixed_size, Image.Resampling.LANCZOS)
```


```python
                    # Map class_id to class name
                    class_name = class_names.get(class_id, "Unknown")

                    # Create a folder for the class if it doesn't exist
                    class_folder = os.path.join(output_folder, class_name)
                    if not os.path.exists(class_folder):
                        os.makedirs(class_folder)

                    # Save the resized image
                    cropped_image_filename = f"{os.path.splitext(image_filename)[0]}_{x_min}_{y_min}.jpg"  # Adjust naming as needed
                    cropped_image_path = os.path.join(class_folder, cropped_image_filename)
                    resized_img.save(cropped_image_path)

                    print(f"Saved resized object to {cropped_image_path}")
        except UnicodeDecodeError as e:
            print(f"Error reading file {label_path}: {e}")
```

Step 3:**cnn model :**

Step 1: Uploading the Dataset

```
[ ] from google.colab import files

    # This will open a dialog to select the file from your device
    uploaded = files.upload()
```

```
Choose Files No file chosen        Upload widget is only available when the cell has been executed in the current browser session. Please rerun this cell to enable.
Saving DR_Model_2_1.zip to DR_Model_2_1.zip
```

Description:

- This code uses the files.upload() function from the Google Colab library to open a file selection dialog, allowing you to upload your dataset files from your local device to the Colab environment.

Step 2: Extracting the Uploaded ZIP File

```
import zipfile
import os

# Replace 'Exudates-3 (3)-new.zip' with the actual name of your uploaded file
zip_file = 'DR_Model_2_1.zip'

# Create a directory to extract the contents
extract_dir = '/content/extracted_exudates'

with zipfile.ZipFile(zip_file, 'r') as zip_ref:
    zip_ref.extractall(extract_dir)

# Navigate to the extracted folder
os.chdir(extract_dir)

# List the contents of the directory to confirm extraction
print("Extracted files:")
os.listdir(extract_dir)
```

```
Extracted files:
['DR_Model_2_1']
```

Description:

- This code extracts the contents of a ZIP file named DR_Model_2_1.zip into a specified directory /content/extracted_exudates.
- It then navigates to this extracted directory and lists the contents to confirm that the extraction was successful.

Step 3: Setting Up Image Data Generators

```python
import tensorflow as tf
from tensorflow.keras.preprocessing.image import ImageDataGenerator
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Conv2D, MaxPooling2D, Flatten, Dense, Dropout
from tensorflow.keras.optimizers import Adam
import matplotlib.pyplot as plt
```

```python
# Define paths
train_dir = '/content/extracted_exudates/DR_Model_2_1/train'
valid_dir = '/content/extracted_exudates/DR_Model_2_1/valid'
test_dir = '/content/extracted_exudates/DR_Model_2_1/test'

# Image data generators
train_datagen = ImageDataGenerator(rescale=1./255)
valid_datagen = ImageDataGenerator(rescale=1./255)
test_datagen = ImageDataGenerator(rescale=1./255)

# Load images
train_generator = train_datagen.flow_from_directory(
    train_dir,
    target_size=(150, 150),
    batch_size=32,
    class_mode='binary'
)

valid_generator = valid_datagen.flow_from_directory(
    valid_dir,
    target_size=(150, 150),
    batch_size=32,
    class_mode='binary'
)

test_generator = test_datagen.flow_from_directory(
    test_dir,
    target_size=(150, 150),
    batch_size=32,
    class_mode='binary'
)
```

```
Found 14265 images belonging to 1 classes.
Found 1593 images belonging to 1 classes.
Found 799 images belonging to 1 classes.
```

Description:

- This step defines the paths to the training, validation, and test datasets.
- It uses ImageDataGenerator to create data generators for each dataset, rescaling pixel values to the range [0, 1].
- The flow_from_directory function loads images from the specified directories, resizes them to 150x150 pixels, and sets up batches for binary classification.

Step 4: Building the Initial CNN Model:

```
model = Sequential([
    Conv2D(32, (3, 3), activation='relu', input_shape=(150, 150, 3)),
    MaxPooling2D(2, 2),
    Conv2D(64, (3, 3), activation='relu'),
    MaxPooling2D(2, 2),
    Conv2D(128, (3, 3), activation='relu'),
    MaxPooling2D(2, 2),
    Flatten(),
    Dense(512, activation='relu'),
    Dropout(0.5),
    Dense(1, activation='sigmoid')
])

model.compile(optimizer=Adam(learning_rate=0.001),
              loss='binary_crossentropy',
              metrics=['accuracy'])

model.summary()
```

Model: "sequential_1"

Layer (type)	Output Shape	Param #
conv2d_6 (Conv2D)	(None, 148, 148, 32)	896
max_pooling2d_6 (MaxPooling2D)	(None, 74, 74, 32)	0
conv2d_7 (Conv2D)	(None, 72, 72, 64)	18,496
max_pooling2d_7 (MaxPooling2D)	(None, 36, 36, 64)	0
conv2d_8 (Conv2D)	(None, 34, 34, 128)	73,856
max_pooling2d_8 (MaxPooling2D)	(None, 17, 17, 128)	0
flatten_1 (Flatten)	(None, 36992)	0

Description:

- This code defines a simple CNN model using the Sequential API from Keras.
- The model consists of three convolutional layers followed by max-pooling layers, a flatten layer, a dense layer with 512 neurons and ReLU activation, a dropout layer for regularization, and an output layer with a sigmoid activation function for binary classification.
- The model is compiled using the Adam optimizer, binary cross-entropy loss function, and accuracy as the metric.
- The model.summary() function prints a summary of the model architecture.

Step 5: Building a More Complex CNN Model with Batch Normalization(this not required)

```
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Conv2D, MaxPooling2D, Flatten, Dense, Dropout, BatchNormalization
from tensorflow.keras.optimizers import Adam

model = Sequential([
    Conv2D(32, (3, 3), activation='relu', input_shape=(150, 150, 3)),
    BatchNormalization(),
    MaxPooling2D(2, 2),

    Conv2D(64, (3, 3), activation='relu'),
    BatchNormalization(),
    MaxPooling2D(2, 2),

    Conv2D(128, (3, 3), activation='relu'),
    BatchNormalization(),
    MaxPooling2D(2, 2),

    Conv2D(256, (3, 3), activation='relu'),
    BatchNormalization(),
    MaxPooling2D(2, 2),

    Conv2D(512, (3, 3), activation='relu'),
    BatchNormalization(),
    MaxPooling2D(2, 2),

    Conv2D(512, (3, 3), activation='relu'),
    BatchNormalization(),
    MaxPooling2D(2, 2),

    Flatten(),
    Dense(1024, activation='relu'),
    Dropout(0.5),

    Dense(512, activation='relu'),
    Dropout(0.5),

    Dense(256, activation='relu'),
    Dropout(0.5),

    Dense(1, activation='sigmoid')
])

model.compile(optimizer=Adam(learning_rate=0.001),
              loss='binary_crossentropy',
              metrics=['accuracy'])

model.summary()
```

Model: "sequential_2"

Description:

- This version of the CNN model includes additional convolutional layers and Batch Normalization layers, which help stabilize and speed up training.
- The model uses max-pooling layers after each convolutional block to downsample the feature maps.
- Additional dense layers are used towards the end with dropout layers in between to reduce overfitting.
- This model architecture is more complex and likely to capture more detailed patterns in the data due to its depth.

Step 6: Extracting Model Architecture Information into a Table:

```
import pandas as pd
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Conv2D, MaxPooling2D, Flatten, Dense, Dropout, BatchNormalization
from tensorflow.keras.optimizers import Adam

# Define the model
model = Sequential([
    Conv2D(32, (3, 3), activation='relu', input_shape=(150, 150, 3)),  # Layer 1
    BatchNormalization(),  # Layer 2
    MaxPooling2D(2, 2),  # Layer 3

    Conv2D(64, (3, 3), activation='relu'),  # Layer 4
    BatchNormalization(),  # Layer 5
    MaxPooling2D(2, 2),  # Layer 6

    Conv2D(128, (3, 3), activation='relu'),  # Layer 7
    BatchNormalization(),  # Layer 8
    MaxPooling2D(2, 2),  # Layer 9

    Conv2D(256, (3, 3), activation='relu'),  # Layer 10
    BatchNormalization(),  # Layer 11
    MaxPooling2D(2, 2),  # Layer 12

    Conv2D(512, (3, 3), activation='relu'),  # Layer 13
    BatchNormalization(),  # Layer 14
    MaxPooling2D(2, 2),  # Layer 15

    Conv2D(512, (3, 3), activation='relu'),  # Layer 16
    BatchNormalization(),  # Layer 17
    MaxPooling2D(2, 2),  # Layer 18

    Flatten(),  # Layer 19
    Dense(1024, activation='relu'),  # Layer 20
    Dropout(0.5),  # Layer 21

    Dense(512, activation='relu'),  # Layer 22
    Dropout(0.5),  # Layer 23

    Dense(256, activation='relu'),  # Layer 24
    Dropout(0.5),  # Layer 25

    Dense(1, activation='sigmoid')  # Output Layer
])

model.compile(optimizer=Adam(learning_rate=0.001),
              loss='binary_crossentropy',
              metrics=['accuracy'])

# Extracting information and creating the table
layer_data = []

for i, layer in enumerate(model.layers):
    config = layer.get_config()
    layer_type = type(layer).__name__
    maps = config['filters'] if 'filters' in config else '-'
    # Correctly access output_shape
    output_shape = layer.output_shape if hasattr(layer, 'output_shape') else None
    output_size = f"{output_shape[1]} x {output_shape[2]}" if output_shape and len(output_shape) > 2 else '-'
    kernel_size = config['kernel_size'] if 'kernel_size' in config else '-'

    layer_data.append({
        'No.': i + 1,
        'Layer Type': layer_type,
        'Maps': maps,
        'Output Size': output_size,
        'Kernel Size': kernel_size
    })

# Create a DataFrame
df = pd.DataFrame(layer_data)
```

	No.	Layer Type	Maps	Output Size	Kernel Size
0	1	Conv2D	32	-	(3, 3)
1	2	BatchNormalization	-	-	-
2	3	MaxPooling2D	-	-	-
3	4	Conv2D	64	-	(3, 3)
4	5	BatchNormalization	-	-	-
5	6	MaxPooling2D	-	-	-
6	7	Conv2D	128	-	(3, 3)
7	8	BatchNormalization	-	-	-
8	9	MaxPooling2D	-	-	-
9	10	Conv2D	256	-	(3, 3)
10	11	BatchNormalization	-	-	-
11	12	MaxPooling2D	-	-	-
12	13	Conv2D	512	-	(3, 3)
13	14	BatchNormalization	-	-	-
14	15	MaxPooling2D	-	-	-
15	16	Conv2D	512	-	(3, 3)
16	17	BatchNormalization	-	-	-
17	18	MaxPooling2D	-	-	-
18	19	Flatten	-	-	-
19	20	Dense	-	-	-
20	21	Dropout	-	-	-
21	22	Dense	-	-	-
22	23	Dropout	-	-	-
23	24	Dense	-	-	-
24	25	Dropout	-	-	-
25	26	Dense	-	-	-

Description:

- This code extracts detailed information about each layer in the model, including layer type, number of filters (for convolutional layers), output size, and kernel size.
- It creates a structured table using pandas, which helps in understanding the model architecture layer by layer.

Step 7: Defining the Final Model with Callbacks for Training:

```
from tensorflow.keras.callbacks import EarlyStopping, ModelCheckpoint

# Define the callbacks with the correct file extension
model_checkpoint = ModelCheckpoint('best_model.keras', save_best_only=True, monitor='val_loss', mode='min')
```

```
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Conv2D, MaxPooling2D, Flatten, Dense, Dropout, BatchNormalization, Input
from tensorflow.keras.optimizers import Adam
from tensorflow.keras.callbacks import EarlyStopping, ModelCheckpoint

# Define the modified model
model = Sequential([
    Input(shape=(150, 150, 3)),  # Define the input layer with Input() for the sequential model
    Conv2D(32, (3, 3), activation='relu', padding='same'),  # Layer 1
    BatchNormalization(),  # Layer 2
    MaxPooling2D(2, 2),  # Layer 3

    Conv2D(64, (3, 3), activation='relu', padding='same'),  # Layer 4
    BatchNormalization(),  # Layer 5
    MaxPooling2D(2, 2),  # Layer 6

    Conv2D(128, (3, 3), activation='relu', padding='same'),  # Layer 7
    BatchNormalization(),  # Layer 8
    MaxPooling2D(2, 2),  # Layer 9

    Conv2D(256, (3, 3), activation='relu', padding='same'),  # Layer 10
    BatchNormalization(),  # Layer 11
    MaxPooling2D(2, 2),  # Layer 12

    Conv2D(512, (3, 3), activation='relu', padding='same'),  # Layer 13
    BatchNormalization(),  # Layer 14
    MaxPooling2D(2, 2),  # Layer 15

    Flatten(),  # Layer 16
    Dense(1024, activation='relu'),  # Layer 17
    Dropout(0.5),  # Layer 18

    Dense(512, activation='relu'),  # Layer 19
    Dropout(0.5),  # Layer 20

    Dense(256, activation='relu'),  # Layer 21
    Dropout(0.5),  # Layer 22

    Dense(1, activation='sigmoid')  # Output Layer
])

model.compile(optimizer=Adam(learning_rate=0.001),
              loss='binary_crossentropy',
              metrics=['accuracy'])

# Define callbacks
early_stopping = EarlyStopping(monitor='val_loss', patience=3, restore_best_weights=True)
model_checkpoint = ModelCheckpoint('best_model.keras', save_best_only=True, monitor='val_loss', mode='min')

# Train the model
history = model.fit(
    train_generator,
    steps_per_epoch=train_generator.samples // train_generator.batch_size,
    validation_data=valid_generator,
    validation_steps=valid_generator.samples // valid_generator.batch_size,
    epochs=11,
    callbacks=[early_stopping, model_checkpoint]
)
```

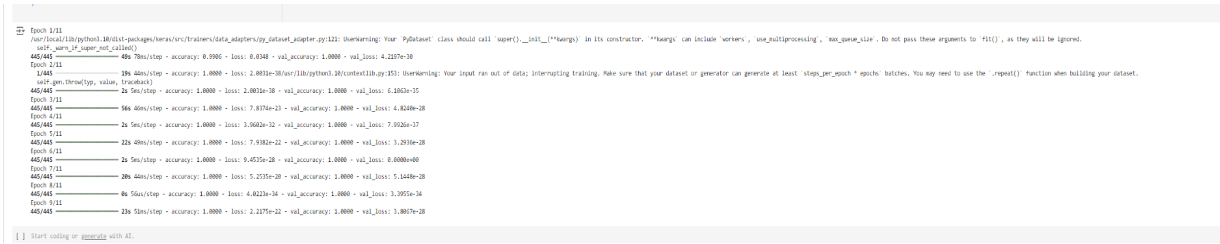
Description:

- This step defines the final CNN model with increased complexity, integrating padding in convolutional layers to maintain spatial dimensions.
- It uses Batch Normalization after each convolutional layer to enhance training performance and stability.
- Dense layers towards the end with dropout regularization improve the model's ability to generalize on unseen data.

Callbacks:

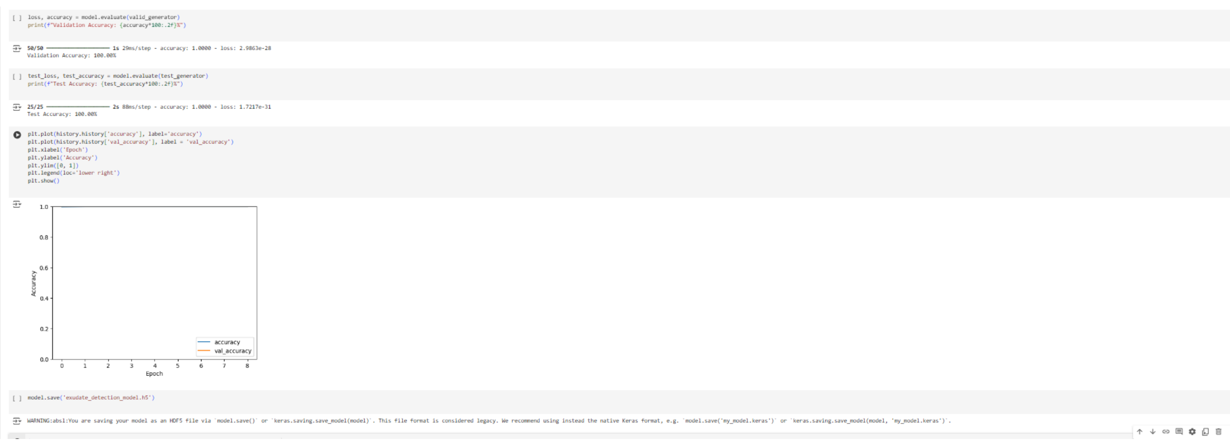
- EarlyStopping monitors the validation loss and stops training if it does not improve for 3 consecutive epochs, restoring the best weights.
- ModelCheckpoint saves the best model based on the lowest validation loss during training.

The model.fit() function trains the model using the specified number of epochs, generators, and callbacks.



```
Epoch 1/11
/usr/local/lib/python3.10/dist-packages/keras/src/trainers/data_adapters/py_dataset_adapter.py:121: UserWarning: Your `PyDataset` class should call `super().__init__(**kwargs)` in its constructor. `**kwargs` can include `workers`, `use_multiprocessing`, `max_queue_size`. Do not pass these arguments to `fit()`, as they will be ignored.
  self._warn_if_super_not_called()
445/445 ━━━━━━━━━━━━━━━━━━━━ 49s 78ms/step - accuracy: 0.9906 - loss: 0.0348 - val_accuracy: 1.0000 - val_loss: 4.2197e-30
Epoch 2/11
  1/445 ━━━━━━━━━━━━━━━━━━━━ 19s 44ms/step - accuracy: 1.0000 - loss: 2.0031e-38/usr/lib/python3.10/contextlib.py:153: UserWarning: Your input ran out of data; interrupting training. Make sure that your dataset or generator can generate at least `steps_per_epoch * epochs` batches. You may need to use the `.repeat()` function when building your dataset.
  self.gen.throw(typ, value, traceback)
445/445 ━━━━━━━━━━━━━━━━━━━━ 2s 5ms/step - accuracy: 1.0000 - loss: 2.0031e-38 - val_accuracy: 1.0000 - val_loss: 6.1063e-35
Epoch 3/11
445/445 ━━━━━━━━━━━━━━━━━━━━ 56s 46ms/step - accuracy: 1.0000 - loss: 7.8374e-23 - val_accuracy: 1.0000 - val_loss: 4.8240e-28
Epoch 4/11
445/445 ━━━━━━━━━━━━━━━━━━━━ 2s 5ms/step - accuracy: 1.0000 - loss: 3.9602e-32 - val_accuracy: 1.0000 - val_loss: 7.9926e-37
Epoch 5/11
445/445 ━━━━━━━━━━━━━━━━━━━━ 22s 49ms/step - accuracy: 1.0000 - loss: 7.9382e-22 - val_accuracy: 1.0000 - val_loss: 3.2936e-28
Epoch 6/11
445/445 ━━━━━━━━━━━━━━━━━━━━ 2s 5ms/step - accuracy: 1.0000 - loss: 9.4535e-28 - val_accuracy: 1.0000 - val_loss: 0.0000e+00
Epoch 7/11
445/445 ━━━━━━━━━━━━━━━━━━━━ 20s 44ms/step - accuracy: 1.0000 - loss: 5.2535e-20 - val_accuracy: 1.0000 - val_loss: 5.1448e-28
Epoch 8/11
445/445 ━━━━━━━━━━━━━━━━━━━━ 0s 56us/step - accuracy: 1.0000 - loss: 4.0223e-34 - val_accuracy: 1.0000 - val_loss: 3.3955e-34
Epoch 9/11
445/445 ━━━━━━━━━━━━━━━━━━━━ 23s 51ms/step - accuracy: 1.0000 - loss: 2.2175e-22 - val_accuracy: 1.0000 - val_loss: 3.8067e-28

[ ] Start coding or generate with AI.
```

Step 9: Visualizing Training and Validation Performance:



```
[ ] loss, accuracy = model.evaluate(valid_generator)
    print(f"Validation Accuracy: {accuracy*100:.2f}%")

50/50 ━━━━━━━━━━━━━━━━━━━━ 1s 29ms/step - accuracy: 1.0000 - loss: 2.9863e-28
Validation Accuracy: 100.00%

[ ] test_loss, test_accuracy = model.evaluate(test_generator)
    print(f"Test Accuracy: {test_accuracy*100:.2f}%")

25/25 ━━━━━━━━━━━━━━━━━━━━ 2s 88ms/step - accuracy: 1.0000 - loss: 1.7217e-31
Test Accuracy: 100.00%

plt.plot(history.history['accuracy'], label='accuracy')
plt.plot(history.history['val_accuracy'], label = 'val_accuracy')
plt.xlabel('Epoch')
plt.ylabel('Accuracy')
plt.ylim([0, 1])
plt.legend(loc='lower right')
plt.show()
```



```
[ ] model.save('exudate_detection_model.h5')

WARNING:absl:You are saving your model as an HDF5 file via `model.save()` or `keras.saving.save_model(model)`. This file format is considered legacy. We recommend using instead the native Keras format, e.g. `model.save('my_model.keras')` or `keras.saving.save_model(model, 'my_model.keras')`.
```

Description:

- This code visualizes the training and validation loss and accuracy over the epochs.
- The plots provide insights into how well the model is learning and whether there are signs of overfitting or underfitting.

Step 10: Evaluating the Model on the Test Set:

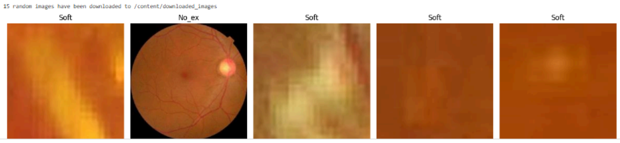
```python
import numpy as np
from tensorflow.keras.preprocessing import image
from PIL import Image
import os

# Directory to save the generated images
save_dir = '/content/random_images'  # Change this path if needed
os.makedirs(save_dir, exist_ok=True)

# Generate 15 random images
for i in range(15):
    # Create a random image array of shape (150, 150, 3)
    random_img_array = np.random.randint(0, 256, (150, 150, 3), dtype=np.uint8)

    # Convert the array to an image
    random_img = Image.fromarray(random_img_array)

    # Save the image
    random_img.save(os.path.join(save_dir, f'random_image_{i+1}.png'))

print(f"15 random images have been saved to {save_dir}")
```

```
15 random images have been saved to /content/random_images
```

```python
import os
import random
from PIL import Image
import matplotlib.pyplot as plt

# Define paths to the directories containing images for each class
soft_dir = '/content/extracted_exudates/DR_Model_2_1/test/output_test/Soft'   # Replace with your path
hard_dir = '/content/extracted_exudates/DR_Model_2_1/test/output_test/Hard'   # Replace with your path
no_ex_dir = '/content/extracted_exudates/DR_Model_2_1/test/output_test/No_ex'  # Replace with your path

# Dictionary mapping class names to their respective directories
class_dirs = {
    'Soft': soft_dir,
    'Hard': hard_dir,
    'No_ex': no_ex_dir
}

# Randomly select a class
random_class = random.choice(list(class_dirs.keys()))

# Get all image files from the selected class directory
image_files = os.listdir(class_dirs[random_class])
# Filter to include only image files (e.g., .jpg, .png)
image_files = [f for f in image_files if f.endswith(('.png', '.jpg', '.jpeg'))]

# Randomly select an image from the class directory
random_image_file = random.choice(image_files)

# Load and display the selected image
img_path = os.path.join(class_dirs[random_class], random_image_file)
img = Image.open(img_path)

# Display the image with the class label
plt.imshow(img)
plt.axis('off')
plt.title(f'Randomly Selected Image - Class: {random_class}')
plt.show()
```

```python
import os
import random
from PIL import Image
import matplotlib.pyplot as plt
import shutil

# Define paths to the directories containing images for each class


# Dictionary mapping class names to their respective directories
class_dirs = {
    'Soft': soft_dir,
    'Hard': hard_dir,
    'No_ex': no_ex_dir
}

# Collect all images from the directories
all_images = []
for class_name, dir_path in class_dirs.items():
    image_files = os.listdir(dir_path)
    image_files = [f for f in image_files if f.endswith(('.png', '.jpg', '.jpeg'))]  # Filter image files
    all_images.extend([(class_name, os.path.join(dir_path, img)) for img in image_files])

# Randomly select 15 images
random_images = random.sample(all_images, min(15, len(all_images)))  # Ensures it doesn't error if <15 images

# Directory to save the downloaded images
download_dir = '/content/downloaded_images'  # Change path if needed
os.makedirs(download_dir, exist_ok=True)

# Copy selected images to the download directory
for class_name, img_path in random_images:
    # Generate a filename to avoid duplicates
    filename = f"{class_name}_{os.path.basename(img_path)}"
    save_path = os.path.join(download_dir, filename)
    shutil.copy(img_path, save_path)

print(f"15 random images have been downloaded to {download_dir}")

# Optionally, display the selected images
plt.figure(figsize=(15, 10))
for i, (class_name, img_path) in enumerate(random_images):
    img = Image.open(img_path)
    plt.subplot(3, 5, i + 1)  # Arrange images in a 3x5 grid
    plt.imshow(img)
    plt.axis('off')
    plt.title(class_name)

plt.tight_layout()
plt.show()
```

```
15 random images have been downloaded to /content/downloaded_images
```



Overview:

This script randomly selects and copies 15 images from three specified directories (representing different classes) to a designated download directory. It also displays the selected images in a grid format using Matplotlib.

Steps:

1. Import Required Libraries:
   - os: Used for directory and file path manipulations.
   - random: Used to randomly select images.
   - PIL (Python Imaging Library): Specifically Image, used for opening and processing images.
   - matplotlib.pyplot: Used for displaying images in a grid layout.
   - shutil: Used for copying files from source to destination.
2. Define Class Directories:
   - class_dirs dictionary maps class names ('Soft', 'Hard', 'No_ex') to their respective directories (soft_dir, hard_dir, no_ex_dir). Each directory contains images corresponding to its class.
3. Collect All Images:
   - The script iterates over each class directory:
     - It lists all files in the directory.
     - Filters files to include only those with image extensions (.png, .jpg, .jpeg).
     - Adds each image path to all_images list along with its class name.
4. Randomly Select 15 Images:
   - random.sample() is used to randomly select up to 15 images from all_images.
   - The min(15, len(all_images)) ensures that the code doesn't throw an error if there are fewer than 15 images available.
5. Create Download Directory:
   - A new directory (download_dir) is created (if it doesn't already exist) to save the selected images.
6. Copy Images to Download Directory:
   - The selected images are copied from their original locations to the download_dir.
   - Each copied image is renamed to include its class name as a prefix to avoid filename conflicts.
7. Display Selected Images:
   - A figure is created with a size of 15x10 inches.
   - A loop iterates over the selected images:
     - Each image is opened using PIL.Image.
     - Displayed in a 3x5 grid using plt.subplot().
     - Each image's class name is used as the title of its subplot.
   - plt.tight_layout() adjusts subplot parameters for better fit.
   - plt.show() renders the grid of images.
8. Output:
   - A message is printed to indicate that the images have been successfully copied to the download_dir.
   - The displayed grid shows up to 15 randomly selected images, arranged by class.

Additional Notes:

- Directory Paths: Ensure that the paths assigned to soft_dir, hard_dir, and no_ex_dir are correct and contain the images you intend to process.
- Image Handling: The script handles standard image formats but can be extended to include more formats if needed.
- Grid Layout: The grid is set to a 3x5 configuration, which works well for displaying 15 images. If the number of images changes, the grid can be adjusted accordingly.

Issues:

Invalid input_shape Argument Warning:

- Issue: A warning occurs when passing an input_shape or input_dim argument directly to layers in Keras Sequential models.

Error Message:
vbnet
Copy code
/usr/local/lib/python3.10/dist-packages/keras/src/layers/convolutional/base_conv.py:107: UserWarning: Do not pass an `input_shape`/`input_dim` argument to a layer. When using Sequential models, prefer using an `Input(shape)` object as the first layer in the model instead.

-
- Solution: Use Input(shape=(...)) as the first layer in the model instead of setting input_shape in subsequent layers.

ModelCheckpoint File Format Error:

- Issue: The ModelCheckpoint callback raises a ValueError because the file path does not end with .keras.

Error Message:
lua
Copy code
ValueError: The filepath provided must end in `.keras` (Keras model format). Received: filepath=best_model.h5

-

Solution: Update the file path to end with .keras, such as:
python
Copy code

model_checkpoint = ModelCheckpoint('best_model.keras', save_best_only=True, monitor='val_loss', mode='min')

-

Missing Image Files in Dataset Folders:

- Issue: The script may fail to find image files in the specified directories, leading to errors when listing or processing files.
- Solution: Ensure that the directories specified for each class (soft_dir, hard_dir, no_ex_dir) contain valid image files with extensions .png, .jpg, or .jpeg.

Insufficient Images for Random Selection:

- Issue: When selecting 15 random images, if fewer than 15 images are available across the classes, the script will adjust but may not meet the expected count.
- Solution: Check the dataset size and adjust the selection logic as necessary. Ensure that each class has a sufficient number of images.

File Copy Errors (Permissions or Path Issues):

- Issue: Errors can occur while copying images to the download directory if there are permissions issues or incorrect paths.
- Solution: Verify that the paths are correct and that you have the necessary permissions to write to the download directory (/content/downloaded_images).

Matplotlib Display Errors:

- Issue: Images may not display correctly if there are issues with the Matplotlib configuration or if PIL fails to open images.
- Solution: Ensure Matplotlib is properly configured in your environment and that the image files are not corrupted.

Class Directory Mapping Errors:

- Issue: Incorrect mapping of class names to directories may lead to misclassification or errors in locating files.
- Solution: Double-check the paths assigned in the class_dirs dictionary to ensure each class name correctly corresponds to its directory.

TensorFlow / Keras Version Incompatibilities:

- Issue: Compatibility issues may arise if the installed versions of TensorFlow and Keras do not support certain functionalities or callback options.
- Solution: Make sure that TensorFlow and Keras are updated to compatible versions. Check for version-specific requirements or deprecated functions.

Memory or Resource Limitations:

- Issue: Running the code on platforms like Google Colab might lead to memory or resource exhaustion, especially with large datasets.
- Solution: Optimize the dataset s

Directory Paths Not Defined:

- The script requires the paths (soft_dir, hard_dir, no_ex_dir) to be defined correctly. If these paths are incorrect or not accessible, the script will fail to locate the images.

Insufficient Number of Images:

- If any class directory contains fewer than 15 images, random.sample() will select only the available images without errors. However, if the overall count of images is critical (e.g., exactly 15 from each class), this can be a problem.

Unsupported Image Formats:

- The script currently supports .png, .jpg, and .jpeg formats. If there are images in other formats (e.g., .bmp, .tiff), they will be ignored unless the script is modified to handle those formats.

File Naming Conflicts:

- While images are renamed with class prefixes to avoid conflicts, there's still a possibility of conflicts if images from different classes share the same filename structure. Further renaming or unique identifier generation may be necessary.

Memory and Performance Constraints:

- Displaying a large number of high-resolution images can be resource-intensive. Ensure that the environment (e.g., Google Colab) has sufficient memory to handle image loading and display without crashing.

Permissions and Access Issues:

- The script needs read access to the source directories and write access to the download directory. Permission errors will cause the script to fail at the copying stage.

Library Dependencies:

- The script relies on several Python libraries (PIL, matplotlib, etc.). Ensure that all necessary libraries are installed and compatible with your environment. Missing libraries or version mismatches can cause runtime errors.

Visualization Layout:

- The current grid layout (3x5) assumes exactly 15 images. If the number of images changes, the layout might need adjustments to avoid awkward spacing or layout issues.

Handling Corrupt or Unsupported Files:

- The script assumes that all files ending in .png, .jpg, or .jpeg are valid images. If a file is corrupt or unsupported by PIL, the script will raise an error. Adding error handling for such cases can make the script more robust.

Reference link:

1.https://roboflow.com/

2.https://roboflow.com/universe -(for dataset)

3.https://www.kaggle.com/datasets

4.https://www.tensorflow.org/tutorials/images/cnn -(for training model)

5.https://www.datacamp.com/tutorial/cnn-tensorflow-python