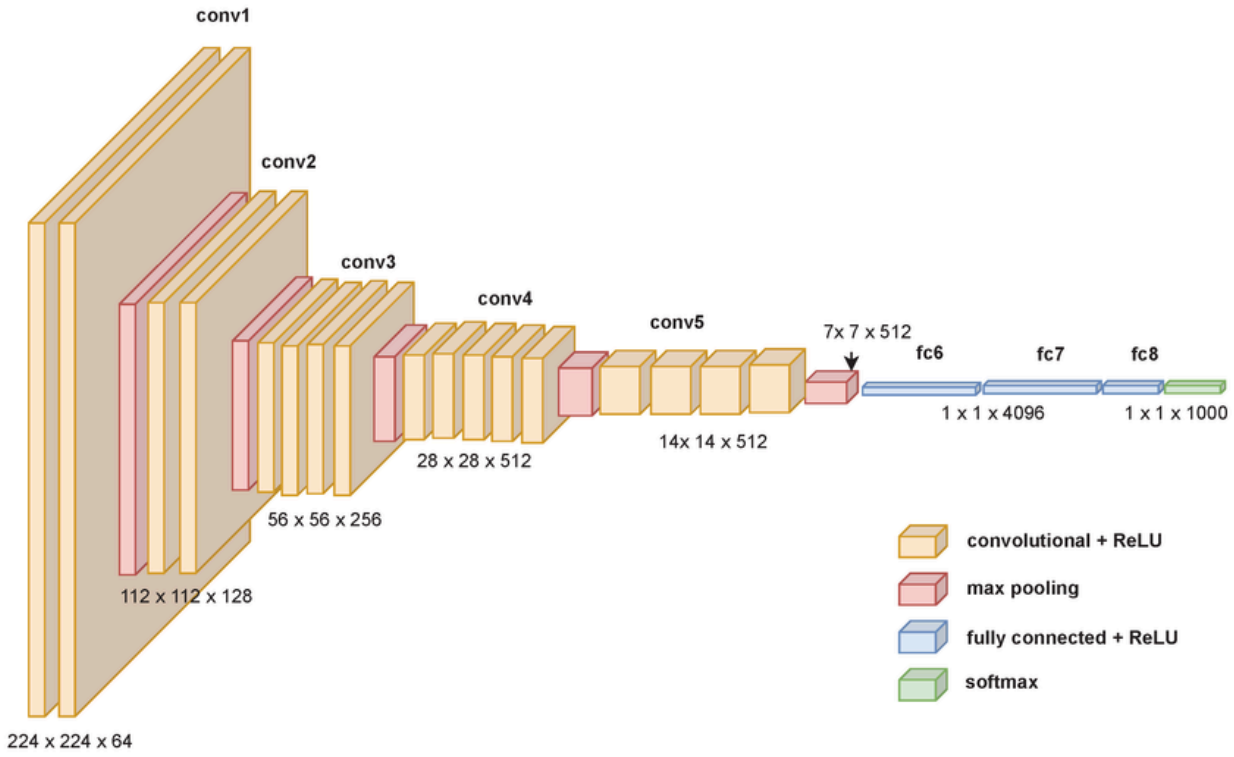
MODEL-2

VGG-19

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.

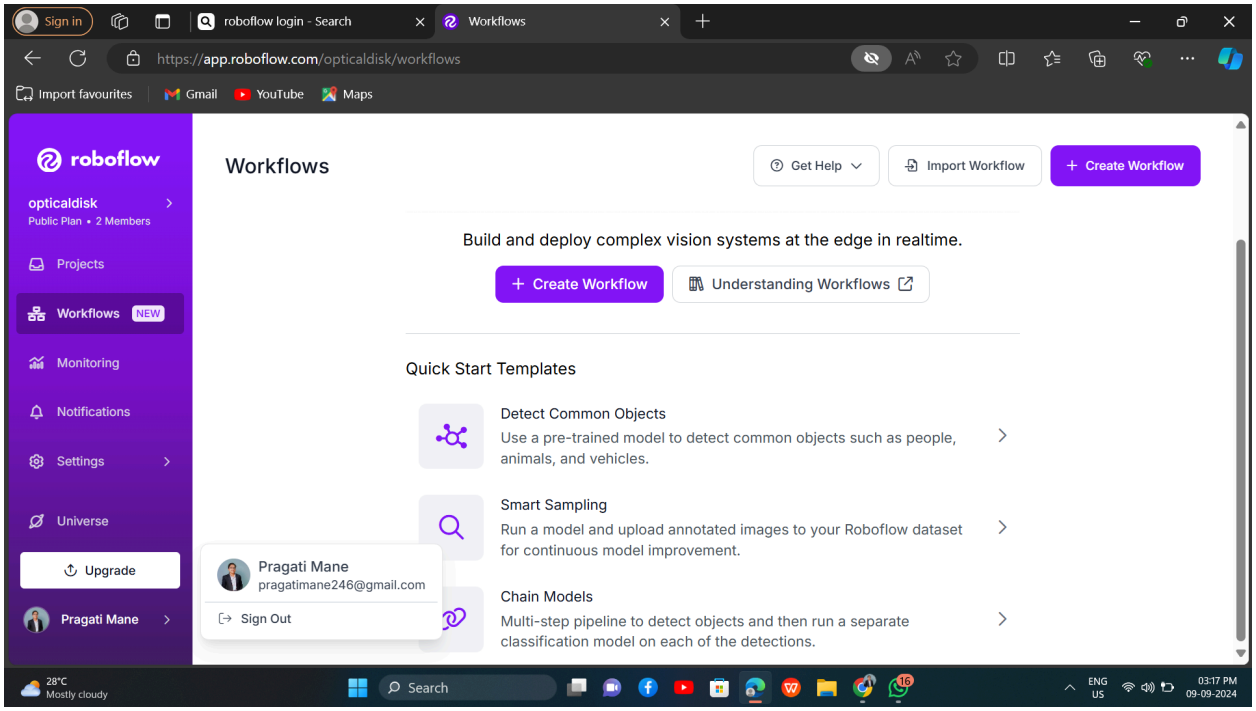
Diagrammatic representation of VGG-19



Roboflow Labeling Dataset

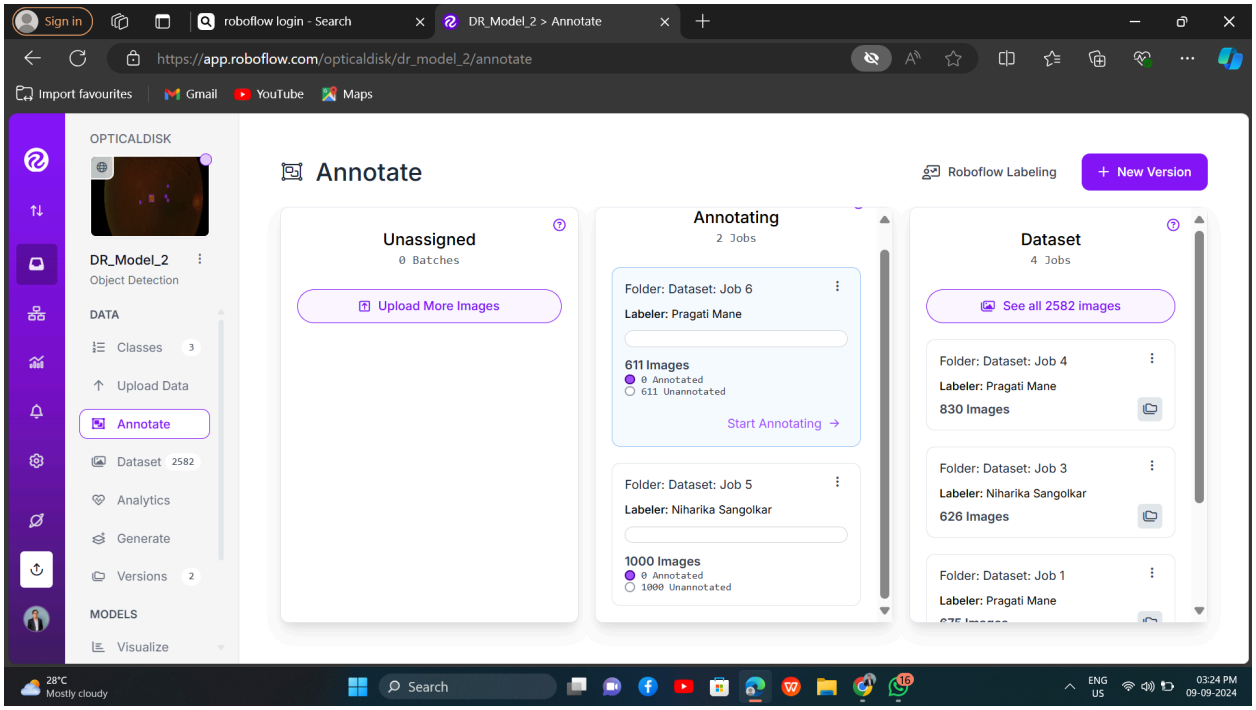
1. Create a Roboflow Account

- Sign up or log in to [Roboflow](https://roboflow.com).
- Once logged in, you will be directed to the Roboflow dashboard.



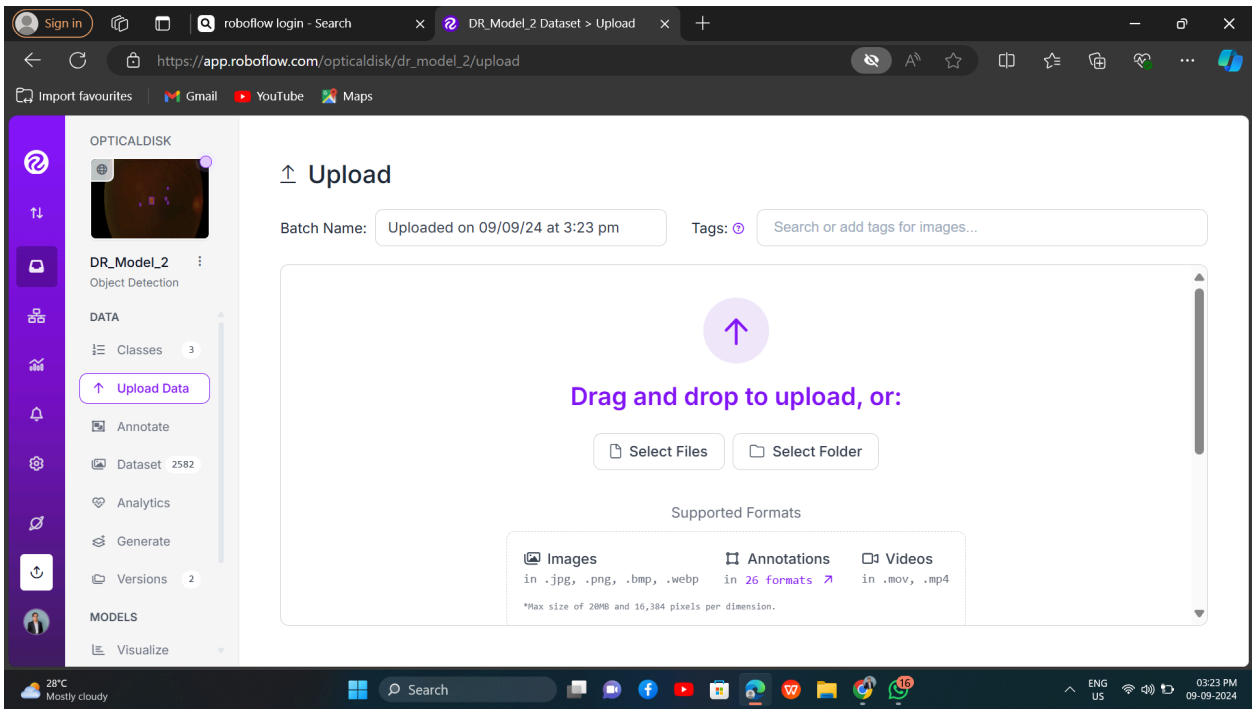
2. Create a New Project

- Click on Create New Project on the dashboard.
- Give your project a name related to diabetic retinopathy (e.g., "Diabetic Retinopathy Detection").
- Choose your task type: Object Detection (since you want to label different types of exudates).
- Select the type of annotation you want: Bounding Box for detecting hard and soft exudates.



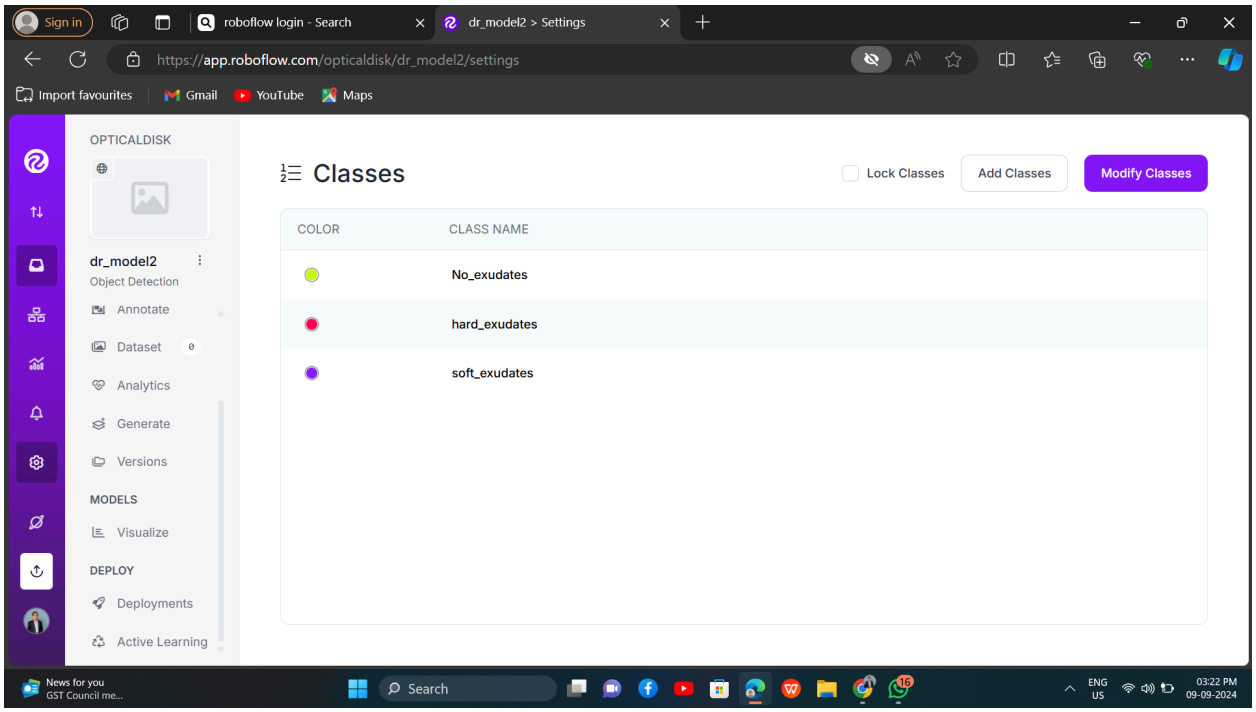
3. Upload Your Images

- After creating the project, you will be prompted to upload images.
- Drag and drop your dataset or select images from your local files. You can upload images for training, testing, and validation here.



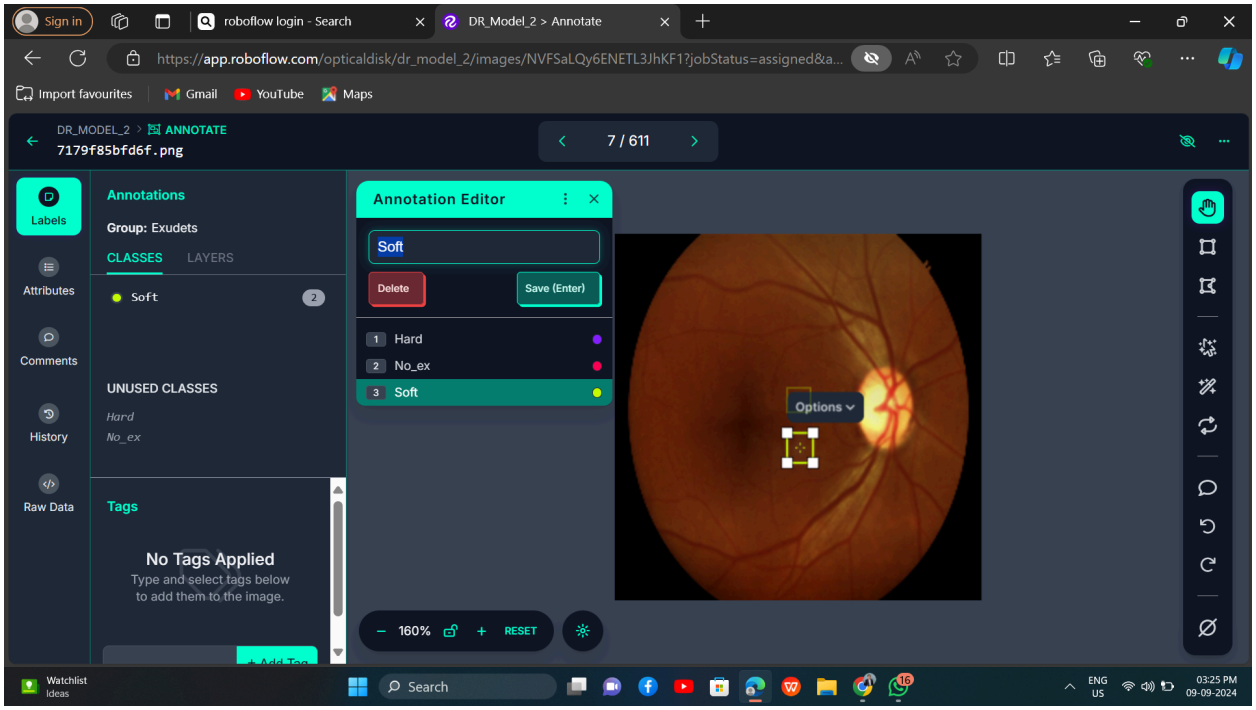
4. Add Classes for Labeling

- After uploading your images, Roboflow will ask you to define your classes.
- Add your classes: Hard Exudates, Soft Exudates, and No Exudates .

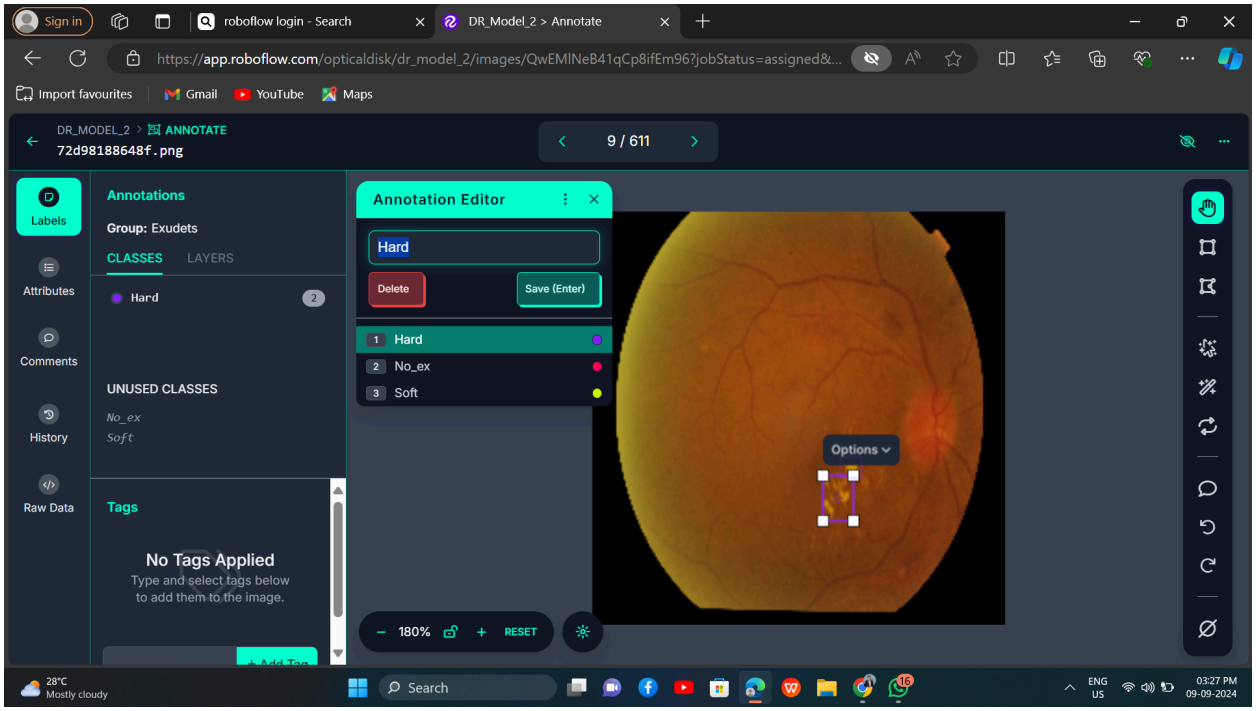


5. Annotate (Label) Your Images

- After your images are uploaded, start annotating them by drawing bounding boxes around the exudates.
- Label each region by selecting the appropriate class: Hard Exudates, Soft Exudates, or No Exudates.
- You can zoom in to accurately label the regions.

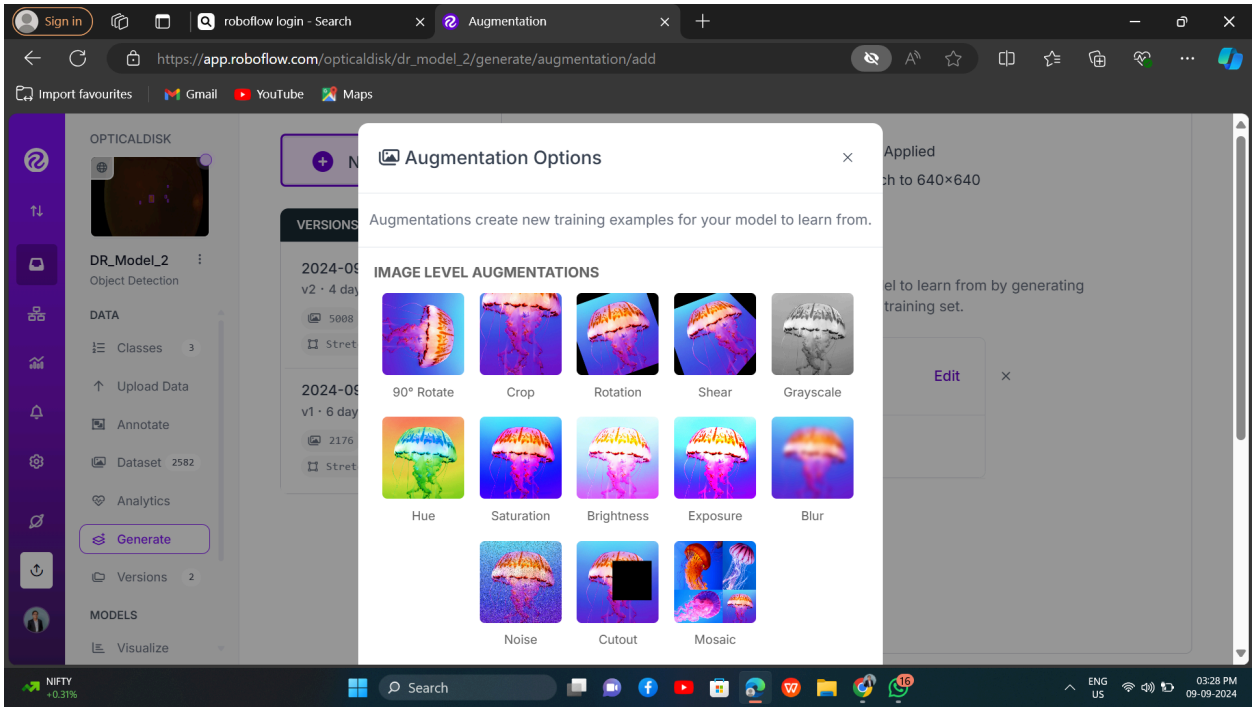


6. Review and Save Annotations
   - Once all your images are annotated, review your work.
   - You can re-adjust bounding boxes if necessary or add additional annotations.
   - After reviewing, save the annotations.



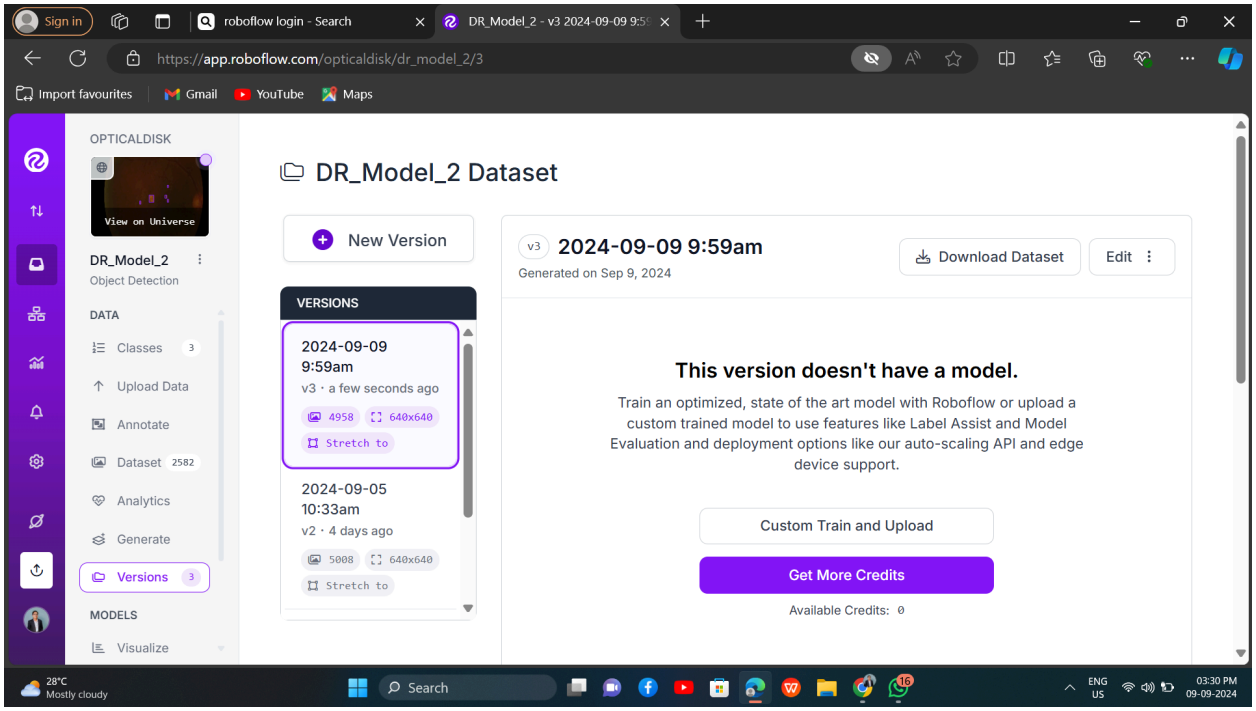
7. Preprocess Data

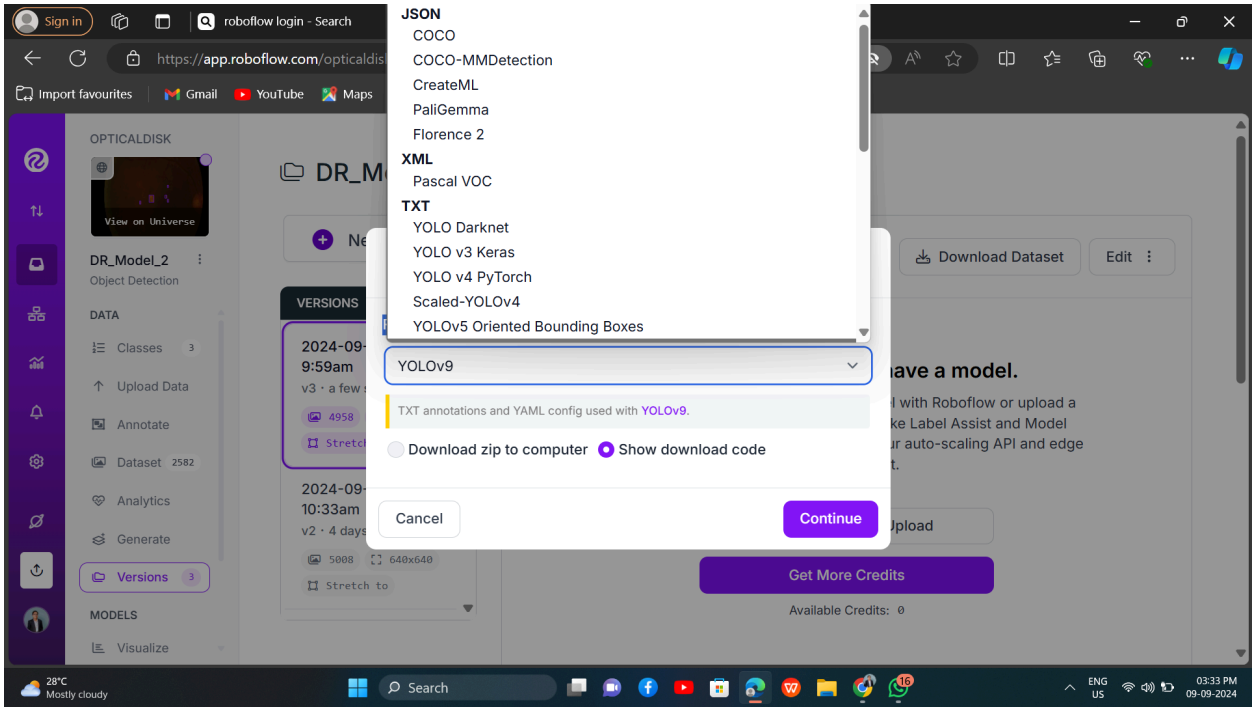
- Roboflow allows you to preprocess the images. You can resize, augment, or normalize your dataset to improve the performance of your model.
- Choose appropriate preprocessing settings based on your project needs.

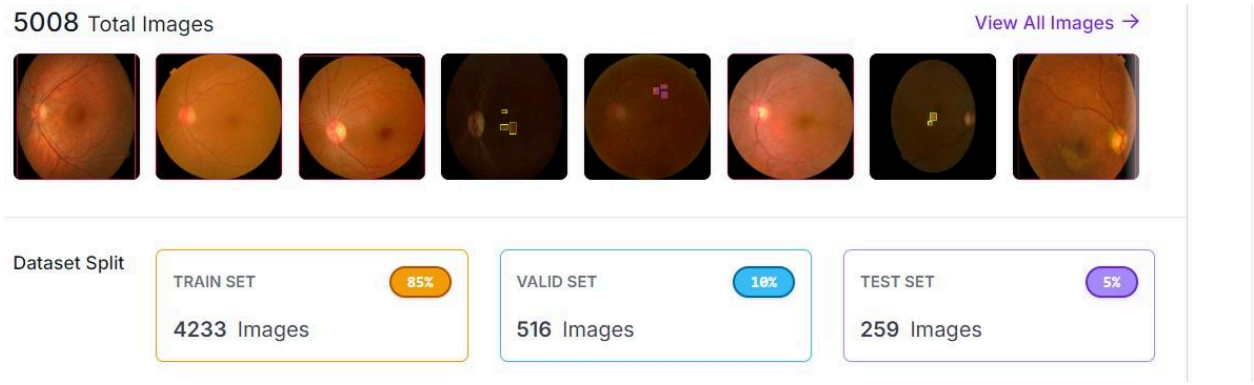


8. Export Dataset

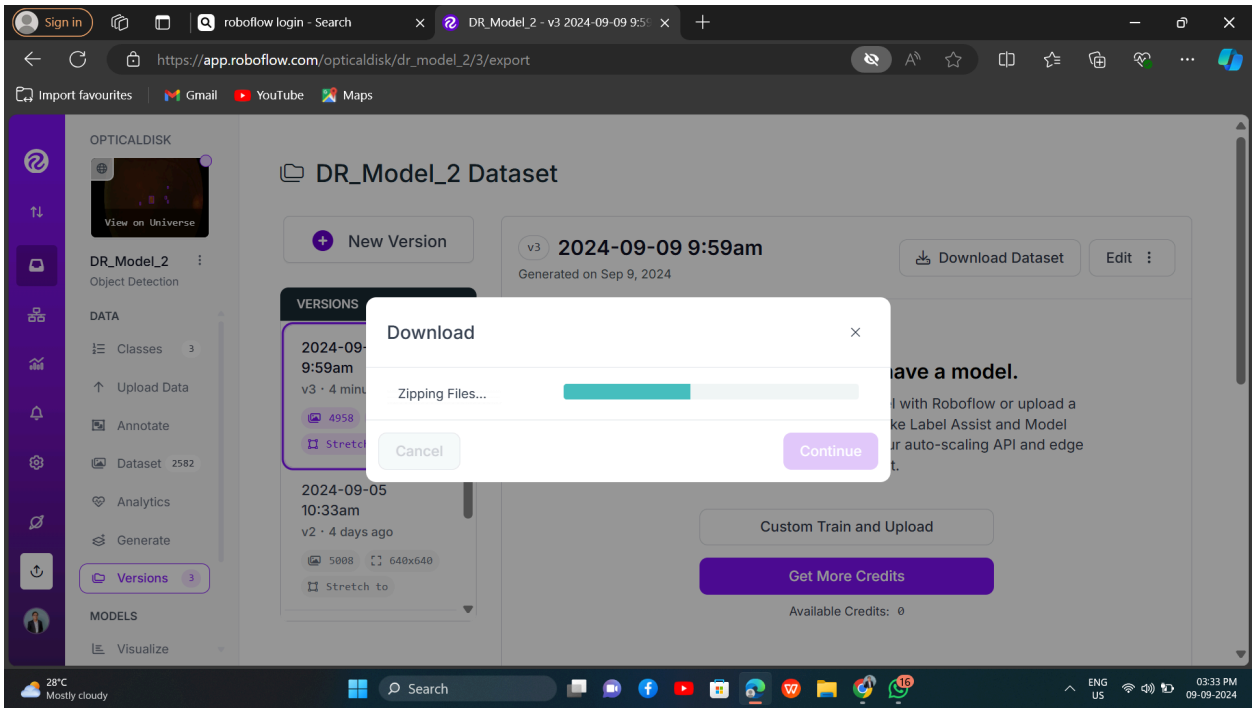
- After completing annotations and preprocessing, you can export the dataset.
- Choose your preferred format, such as TFRecord, COCO JSON, YOLOv5, or others.
- Download the dataset, which includes the images and the respective annotation files.


roboflow login - Search
DR_Model_2 - v3 2024-09-09 9:5
https://app.roboflow.com/opticaldisk/dr_model_2/3
Import favourites
Gmail
YouTube
Maps
OPTICALDISK
View on Universe
DR_Model_2
Object Detection
DATA
Classes 3
Upload Data
Annotate
Dataset 2582
Analytics
Generate
Versions 3
MODELS
Visualize
DR_Model_2 Dataset
New Version
VERSIONS
2024-09-09 9:59am
v3 · a few seconds ago
4958 640x640
Stretch to
2024-09-05 10:33am
v2 · 4 days ago
5008 640x640
Stretch to
v3
2024-09-09 9:59am
Generated on Sep 9, 2024
Download Dataset
Edit
This version doesn't have a model.
Train an optimized, state of the art model with Roboflow or upload a custom trained model to use features like Label Assist and Model Evaluation and deployment options like our auto-scaling API and edge device support.
Custom Train and Upload
Get More Credits
Available Credits: 0
28°C
Mostly cloudy
Search
ENG
US
03:30 PM
09-09-2024


JSON
COCO
COCO-MMDetection
CreateML
PaliGemma
Florence 2
XML
Pascal VOC
TXT
YOLO Darknet
YOLO v3 Keras
YOLO v4 PyTorch
Scaled-YOLOv4
YOLOv5 Oriented Bounding Boxes
YOLOv9
TXT annotations and YAML config used with YOLOv9.
Download zip to computer
Show download code
Cancel
Continue
Download Dataset
Edit
Get More Credits
Available Credits: 0
28°C
Mostly cloudy
Search
ENG
US
03:33 PM
09-09-2024



9. Download Labeled Data
   - Once your dataset is exported, download it to your local machine.
   - You will receive a zipped folder containing the images and annotation files in your chosen format.



VGG-19 Model for Exudate Detection

Code :

```
from google.colab import files

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

Output :

```
Choose Files DR_Model_2_1.zip
• DR_Model_2_1.zip(application/x-zip-compressed) - 50683926 bytes, last modified: 9/5/2024 - 100% done
Saving DR_Model_2_1.zip to DR_Model_2_1.zip
```

Line 1: File Upload and Extraction

python
Copy code
`from google.colab import files`

Explanation:

- This line imports the `files` module from the Google Colab library, which provides the ability to upload files directly from your local computer into the Colab environment.
- It is useful for bringing datasets or any other files necessary for executing code in Google Colab.

Line 2: Extraction

python
Copy code
`uploaded = files.upload()`

Explanation:

- This command opens a file dialog box allowing users to upload files.
- The function `files.upload()` returns a dictionary where the keys are the filenames and the values are the file data.
- In this case, the variable `uploaded` will store the uploaded files, so you can access and process them within the notebook.

Code :

```python
import zipfile
import os

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

Output :

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

Line 1: Import the zipfile module

python
Copy code
import zipfile

- **Explanation:**
  - **import zipfile**: This line imports the zipfile module, which is used for creating, reading, writing, and extracting ZIP archives. This module provides the functionality to handle ZIP files in Python.

Line 2: Import the os module

python
Copy code
import os

- **Explanation:**
  - **import os**: This line imports the os module, which provides a way to interact with the operating system. It includes functions for file and directory operations, such as changing the working directory and listing directory contents.

Line 4: Set the path to the ZIP file

python
Copy code
```
zip_file = '/content/DR_Model_2_1.zip'
```

- **Explanation:**
  - **zip_file = '/content/DR_Model_2_1.zip'**: This line defines the path to the ZIP file you want to extract. You should replace this path with the actual path of your ZIP file if it differs.

Line 6: Define the directory for extraction

python
Copy code
```
extract_dir = '/content/extracted_exudates'
```

- **Explanation:**
  - **extract_dir = '/content/extracted_exudates'**: This line specifies the directory where the contents of the ZIP file will be extracted. If this directory does not already exist, it will be created during the extraction process.

Line 8: Open the ZIP file and extract its contents

python
Copy code
```
with zipfile.ZipFile(zip_file, 'r') as zip_ref:
```

- **Explanation:**
  - **with zipfile.ZipFile(zip_file, 'r') as zip_ref:**: This line opens the specified ZIP file in read mode and assigns it to the variable zip_ref. The with statement ensures that the file is properly closed after the extraction is completed.

Line 9: Extract all files from the ZIP archive

python
Copy code

```
zip_ref.extractall(extract_dir)
```

- **Explanation:**
  - **zip_ref.extractall(extract_dir)**: This line extracts all the files and directories from the ZIP file into the specified directory (extract_dir). This creates a copy of the files from the ZIP archive in the chosen directory.

Line 11: Change the current working directory to the extraction directory

python
Copy code
```
os.chdir(extract_dir)
```

- **Explanation:**
  - **os.chdir(extract_dir)**: This line changes the current working directory to extract_dir. After this operation, any file operations (like reading or writing files) will be relative to this directory.

Line 13: Print a message indicating the listing of files

python
Copy code
```
print("Extracted files:")
```

- **Explanation:**
  - **print("Extracted files:")**: This line prints a message to indicate that the contents of the extraction directory will be listed.

Line 14: List and print the contents of the extraction directory

python
Copy code
```
os.listdir(extract_dir)
```

- **Explanation:**
  - **os.listdir(extract_dir)**: This line lists all files and directories within the extract_dir directory and prints them to confirm that the extraction was successful.

Code :

```
from tensorflow.keras.preprocessing.image import ImageDataGenerator

# Create ImageDataGenerator instances for training, validation, and test datasets
train_datagen = ImageDataGenerator(rescale=1./255)
valid_datagen = ImageDataGenerator(rescale=1./255)
test_datagen = ImageDataGenerator(rescale=1./255)

# Load images with specified parameters
train_generator = train_datagen.flow_from_directory(
    train_dir,
    target_size=(224, 224),  # Resize images to 224x224 for VGG19
    batch_size=32,
    class_mode='binary'  # Binary classification
)

valid_generator = valid_datagen.flow_from_directory(
    valid_dir,
    target_size=(224, 224),
    batch_size=32,
    class_mode='binary'
)
```

```
test_generator = test_datagen.flow_from_directory(
    test_dir,
    target_size=(224, 224),
    batch_size=32,
    class_mode='binary'
)
```

Output :

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

Line 1: Import `ImageDataGenerator`

python
Copy code

```
from tensorflow.keras.preprocessing.image import ImageDataGenerator
```

- **Explanation:**
  This line imports the `ImageDataGenerator` class from the `tensorflow.keras.preprocessing.image` module. `ImageDataGenerator` is used for generating batches of image data with real-time data augmentation. It helps in rescaling, transforming, and augmenting images for training, validation, and testing purposes.

Line 2-4: Create `ImageDataGenerator` instances

python
Copy code

```
train_datagen = ImageDataGenerator(rescale=1./255)
valid_datagen = ImageDataGenerator(rescale=1./255)
test_datagen = ImageDataGenerator(rescale=1./255)
```

- **Explanation:**
  These lines create instances of `ImageDataGenerator` for training, validation, and test datasets. The `rescale=1./255` argument normalizes the pixel values of the images from a range of [0, 255] to [0, 1], which is standard practice for deep learning models to enhance convergence during training.

Line 5: Load images from the training directory

python
Copy code

```
train_generator = train_datagen.flow_from_directory(
    train_dir,
    target_size=(224, 224),  # Resize images to 224x224 for VGG19
    batch_size=32,
    class_mode='binary'  # Binary classification
)
```

- **Explanation:**
  This line sets up the `train_generator` to load images from the `train_dir` directory. The images are resized to 224x224 pixels, which is a standard input size for the VGG19 model. `batch_size=32` means the generator will yield batches of 32 images at a time. `class_mode='binary'` specifies that the labels are binary, i.e., two classes.

Line 6: Load images from the validation directory

python
Copy code

```
valid_generator = valid_datagen.flow_from_directory(
    valid_dir,
    target_size=(224, 224),
    batch_size=32,
    class_mode='binary'
}
```

- **Explanation:**
  This line sets up the `valid_generator` to load images from the `valid_dir` directory. The images are resized to 224x224 pixels, with a batch size of 32. `class_mode='binary'` indicates that the validation set is used for binary classification.

Line 7: Load images from the test directory

python
Copy code

```
test_generator = test_datagen.flow_from_directory(
    test_dir,
    target_size=(224, 224),
    batch_size=32,
    class_mode='binary'
)
```

- **Explanation:**
  This line sets up the `test_generator` to load images from the `test_dir` directory. The images are resized to 224x224 pixels, with a batch size of 32. `class_mode='binary'` specifies that the test set is used for binary classification.

Code :

```
from tensorflow.keras.applications import VGG19
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense, Dropout, Flatten, BatchNormalization

# Load VGG19 model pre-trained on ImageNet, excluding the top classification layers
base_model = VGG19(weights='imagenet', include_top=False, input_shape=(224, 224, 3))

# Freeze all layers in the base model
for layer in base_model.layers:
    layer.trainable = False
```

Output :

```
Downloading data from https://storage.googleapis.com/tensorflow/keras-applications/vgg19/vgg19_weights_tf_dim_ordering_tf_kernels_n
80134624/80134624 ━━━━━━━━━━━━━━━━━━━━ 1s 0us/step
```

Line 1: Import VGG19 model

python
Copy code

```
from tensorflow.keras.applications import VGG19
```

- **Explanation:**
  This line imports the `VGG19` model from `tensorflow.keras.applications`. `VGG19` is a deep convolutional neural network with 19 layers, known for its effectiveness in image classification tasks. It comes pre-trained on the ImageNet dataset, which contains millions of labeled images.

Line 2: Import necessary layers and model class

python
Copy code

```
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense, Dropout, Flatten,
BatchNormalization
```

Explanation:
These lines import the `Sequential` model class and several layers from `tensorflow.keras.layers`:

- **`Sequential`**: This class allows you to build a model layer-by-layer in a sequential manner.
- **`Dense`**: A fully connected neural network layer commonly used in classification tasks.
- **`Dropout`**: A regularization technique that randomly sets a fraction of input units to 0 during training to prevent overfitting.
- **`Flatten`**: Converts the multi-dimensional output of the convolutional layers into a one-dimensional array before passing it to fully connected layers.
- **`BatchNormalization`**: A layer that normalizes the activations of the previous layer, helping to speed up training and improve model stability.

Line 3: Load VGG19 model pre-trained on ImageNet, excluding the top classification layers

python
Copy code

```
base_model = VGG19(weights='imagenet', include_top=False,
input_shape=(224, 224, 3))
```

Explanation:
This line initializes the `VGG19` model with the following parameters:

- `weights='imagenet'`: Loads the weights of the VGG19 model pre-trained on the ImageNet dataset, allowing the model to leverage learned features from this large, diverse dataset.
- `include_top=False`: Excludes the top classification layers (fully connected layers) of VGG19, as these are specific to the 1000 classes in ImageNet. This allows you to replace them with layers suited for your own classification task.
- `input_shape=(224, 224, 3)`: Specifies the input shape of the images (224x224 pixels with 3 color channels: RGB), which is the required input size for VGG19.

Line 4-5: Freeze all layers in the base model

python
Copy code

```
for layer in base_model.layers:
    layer.trainable = False
```

- **Explanation:**
  This loop iterates over all layers of the `base_model` and sets `layer.trainable = False`. This action freezes the weights of all layers in the VGG19 model, preventing them from being updated during training. This is done to retain the valuable features learned by VGG19 on ImageNet and to focus the training on the new layers that you will add later. This approach is commonly used in transfer learning to leverage pre-trained models effectively.

Code :

```
# Initialize the Sequential model
model = Sequential()

# Add the base VGG19 model
model.add(base_model)

# Flatten the output from the base model
model.add(Flatten())

# Add custom layers to reach a total of 20 layers
# Including Flatten layer, Dense layers, Batch Normalization, and Dropout
model.add(Dense(512, activation='relu'))      # Layer 1
model.add(BatchNormalization())              # Layer 2
model.add(Dropout(0.5))                      # Layer 3
model.add(Dense(256, activation='relu'))     # Layer 4
model.add(BatchNormalization())              # Layer 5
model.add(Dropout(0.5))                      # Layer 6
model.add(Dense(128, activation='relu'))     # Layer 7
model.add(BatchNormalization())              # Layer 8
model.add(Dense(128, activation='relu'))     # Layer 9
```

```
model.add(Dense(128, activation='relu'))     # Layer 9
model.add(BatchNormalization())              # Layer 10
model.add(Dropout(0.5))                      # Layer 11
model.add(Dense(64, activation='relu'))      # Layer 12
model.add(BatchNormalization())              # Layer 13
model.add(Dense(64, activation='relu'))      # Layer 14
model.add(BatchNormalization())              # Layer 15
model.add(Dropout(0.5))                      # Layer 16
model.add(Dense(32, activation='relu'))      # Layer 17
model.add(BatchNormalization())              # Layer 18
model.add(Dense(16, activation='relu'))      # Layer 19
model.add(BatchNormalization())              # Layer 20
model.add(Dense(1, activation='sigmoid'))    # Output layer (21st layer if counting outputs separately)
```

Line 1: Initialize the Sequential model

python
Copy code
`model = Sequential()`

- **Explanation:**
  This line initializes a `Sequential` model, which is a linear stack of layers. The `Sequential` class allows you to add layers one after another to build a neural network architecture in a step-by-step manner.

Line 2: Add the base VGG19 model

python
Copy code
`model.add(base_model)`

- **Explanation:**
  This line adds the previously loaded `VGG19` model (stored in `base_model`) to the `Sequential` model. The `VGG19` model acts as the feature extractor, with its weights frozen to prevent training, allowing you to focus on training the added custom layers.

Line 3: Flatten the output from the base model

python
Copy code
`model.add(Flatten())`

- **Explanation:**
  This line adds a `Flatten` layer to the model, which converts the multi-dimensional output of the

`VGG19` model into a one-dimensional array. This transformation is necessary to connect the convolutional layers of VGG19 to fully connected (dense) layers.

Line 4-22: Add custom layers to reach a total of 20 layers (excluding the output layer)

Layer 1: Dense layer with 512 units
python
Copy code
```
model.add(Dense(512, activation='relu'))
```

- **Explanation:**
  Adds a fully connected `Dense` layer with 512 neurons and a ReLU activation function, which introduces non-linearity to the model. This layer helps in learning complex patterns from the flattened feature map.

Layer 2: Batch Normalization
python
Copy code
```
model.add(BatchNormalization())
```

- **Explanation:**
  Adds a `BatchNormalization` layer, which normalizes the output of the previous layer. This helps stabilize and speed up the training process by reducing internal covariate shift.

Layer 3: Dropout layer with 50% dropout rate
python
Copy code
```
model.add(Dropout(0.5))
```

- **Explanation:**
  Adds a `Dropout` layer with a 50% dropout rate, which randomly sets 50% of the inputs to zero during each forward pass in training, helping to reduce overfitting.

Layer 4: Dense layer with 256 units
python
Copy code
```
model.add(Dense(256, activation='relu'))
```

- **Explanation:**
  Adds a `Dense` layer with 256 neurons and a ReLU activation function to further learn patterns from the data.

Layer 5: Batch Normalization
python
Copy code
```
model.add(BatchNormalization())
```

- **Explanation:**
  Adds another `BatchNormalization` layer to normalize the output of the previous layer, improving training speed and stability.

Layer 6: Dropout layer with 50% dropout rate
python
Copy code
```
model.add(Dropout(0.5))
```

- **Explanation:**
  Adds a `Dropout` layer with a 50% dropout rate to reduce overfitting by randomly dropping connections.

Layer 7: Dense layer with 128 units
python
Copy code
```
model.add(Dense(128, activation='relu'))
```

- **Explanation:**
  Adds a `Dense` layer with 128 neurons and a ReLU activation function to continue learning from the previous layer's features.

Layer 8: Batch Normalization
python
Copy code
```
model.add(BatchNormalization())
```

- **Explanation:**
  Adds another `BatchNormalization` layer to normalize the activations.

Layer 9: Dense layer with 128 units
python
Copy code
```
model.add(Dense(128, activation='relu'))
```

- **Explanation:**
  Adds another `Dense` layer with 128 neurons to further learn from the data.

Layer 10: Batch Normalization
python
Copy code
```
model.add(BatchNormalization())
```

- **Explanation:**
  Adds another `BatchNormalization` layer for normalization of activations.

Layer 11: Dropout layer with 50% dropout rate
python
Copy code
```
model.add(Dropout(0.5))
```

- **Explanation:**
  Adds a `Dropout` layer with a 50% dropout rate to prevent overfitting.

Layer 12: Dense layer with 64 units
python
Copy code
```
model.add(Dense(64, activation='relu'))
```

- **Explanation:**
  Adds a `Dense` layer with 64 neurons and ReLU activation for further feature learning.

Layer 13: Batch Normalization
python
Copy code
```
model.add(BatchNormalization())
```

- **Explanation:**
  Adds a `BatchNormalization` layer to normalize activations.

Layer 14: Dense layer with 64 units
python
Copy code
```
model.add(Dense(64, activation='relu'))
```

- **Explanation:**
  Adds another `Dense` layer with 64 neurons for continued feature extraction.

Layer 15: Batch Normalization
python
Copy code
```
model.add(BatchNormalization())
```

- **Explanation:**
  Adds another `BatchNormalization` layer to stabilize the learning process.

Layer 16: Dropout layer with 50% dropout rate
python
Copy code
```
model.add(Dropout(0.5))
```

- **Explanation:**
  Adds another `Dropout` layer to mitigate overfitting.

Layer 17: Dense layer with 32 units
python
Copy code
```
model.add(Dense(32, activation='relu'))
```

- **Explanation:**
  Adds a `Dense` layer with 32 neurons, learning from the preceding layers.

Layer 18: Batch Normalization
python
Copy code
```
model.add(BatchNormalization())
```

- **Explanation:**
  Adds another `BatchNormalization` layer to normalize the features.

Layer 19: Dense layer with 16 units
python
Copy code
```
model.add(Dense(16, activation='relu'))
```

- **Explanation:**
  Adds a `Dense` layer with 16 neurons, further refining feature learning.

Layer 20: Batch Normalization
python
Copy code
```
model.add(BatchNormalization())
```

- **Explanation:**
  Adds the final `BatchNormalization` layer to ensure normalized outputs.

Output Layer (21st layer if counting separately): Dense layer with 1 unit for binary classification

python
Copy code
```
model.add(Dense(1, activation='sigmoid'))
```

- **Explanation:**
  Adds the output layer, a `Dense` layer with 1 neuron and a sigmoid activation function, which outputs a probability between 0 and 1 for binary classification tasks.

Code :

```
from tensorflow.keras.optimizers import Adam

# Compile the model with Adam optimizer and binary crossentropy loss
model.compile(optimizer=Adam(learning_rate=0.001),
              loss='binary_crossentropy',
              metrics=['accuracy'])
```

Line 1: Import the Adam optimizer

python
Copy code
```
from tensorflow.keras.optimizers import Adam
```

- **Explanation:**
  This line imports the `Adam` optimizer from `tensorflow.keras.optimizers`. Adam (Adaptive Moment Estimation) is a popular optimization algorithm in deep learning that combines the advantages of two other extensions of stochastic gradient descent: AdaGrad and RMSProp. It adapts the learning rate for each parameter, making it efficient and effective for training deep neural networks.

Line 2-4: Compile the model with Adam optimizer and binary crossentropy loss

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

Explanation:

- **`optimizer=Adam(learning_rate=0.001)`**:
  This sets the optimizer for the model to Adam with a learning rate of `0.001`. The learning rate controls how much the model weights are adjusted with respect to the loss gradient. A smaller learning rate like `0.001` helps in fine-tuning the model adjustments without making drastic changes that could destabilize the learning process.
- **`loss='binary_crossentropy'`**:
  This specifies the loss function as binary crossentropy, which is commonly used for binary classification problems. Binary crossentropy measures the dissimilarity between the predicted probabilities and the actual binary labels (0 or 1). It helps the model minimize the difference between predicted outputs and true values.
- **`metrics=['accuracy']`**:
  This specifies the metric used to evaluate the model's performance during training and testing, in this case, accuracy. Accuracy measures the proportion of correctly classified instances out of the total instances and is a key metric for binary classification tasks.

Code :

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

# Define EarlyStopping to stop training when validation loss stops improving
early_stopping = EarlyStopping(monitor='val_loss', patience=3, restore_best_weights=True)

# Define ModelCheckpoint to save the best model during training
model_checkpoint = ModelCheckpoint('best_vgg19_model.keras', save_best_only=True, monitor='val_loss', mode='min')
```

Line 1: Import EarlyStopping and ModelCheckpoint callbacks

python
Copy code

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

- **Explanation:**
  This line imports the `EarlyStopping` and `ModelCheckpoint` callbacks from `tensorflow.keras.callbacks`. Callbacks are functions or classes that can be used during training to monitor various aspects of the model, such as performance and improvements, and to take actions like stopping training early or saving the model.

Line 2-3: Define EarlyStopping to stop training when validation loss stops improving

python
Copy code

```
early_stopping = EarlyStopping(monitor='val_loss', patience=3,
restore_best_weights=True)
```

Explanation:
This line creates an instance of the `EarlyStopping` callback with the following parameters:

- **`monitor='val_loss'`**:
  Specifies that the callback should monitor the validation loss during training. Validation loss is a key indicator of how well the model generalizes to unseen data.
- **`patience=3`**:
  Sets the patience parameter to 3, meaning the training will stop if the validation loss does not improve for 3 consecutive epochs. This helps prevent overfitting and saves time by halting training when further improvements are unlikely.
- **`restore_best_weights=True`**:
  Ensures that the model weights are reverted to the best weights observed during training when early stopping occurs. This way, the model retains the best performance even if training stops early due to the validation loss plateauing.

Line 4-5: Define ModelCheckpoint to save the best model during training

python
Copy code

```
model_checkpoint = ModelCheckpoint('best_vgg19_model.keras',
save_best_only=True, monitor='val_loss', mode='min')
```

Explanation:
This line creates an instance of the `ModelCheckpoint` callback with the following parameters:

- **`'best_vgg19_model.keras'`**:
  Specifies the file path where the best model will be saved during training. The file will be named `best_vgg19_model.keras`.
- **`save_best_only=True`**:
  Instructs the callback to save the model only when the monitored metric improves. This prevents overwriting the best model with worse-performing versions.
- **`monitor='val_loss'`**:
  Specifies that the callback should monitor the validation loss to determine the best model.
- **`mode='min'`**:
  Indicates that the model should be saved when the validation loss reaches its minimum value, as lower validation loss is preferred.

Code :

```
# 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,  # Number of epochs
    callbacks=[early_stopping, model_checkpoint]  # Use callbacks
)
```

Output :

```
Epoch 1/11
/usr/local/lib/python3.10/dist-packages/keras/src/trainers/data_adapters/py_dataset_adapter.py:121: UserWarning: Your `PyDataset` c
  self._warn_if_super_not_called()
445/445 ━━━━━━━━━━━━━━━━━━━━ 129s 231ms/step - accuracy: 0.7879 - loss: 0.5258 - val_accuracy: 1.0000 - val_loss: 0.0512
Epoch 2/11
  1/445 ━━━━━━━━━━━━━━━━━━━━ 1:19 179ms/step - accuracy: 1.0000 - loss: 0.0503/usr/lib/python3.10/contextlib.py:153: UserWarning: Y
  self.gen.throw(typ, value, traceback)
445/445 ━━━━━━━━━━━━━━━━━━━━ 1s 2ms/step - accuracy: 1.0000 - loss: 0.0503 - val_accuracy: 1.0000 - val_loss: 0.0525
Epoch 3/11
445/445 ━━━━━━━━━━━━━━━━━━━━ 106s 196ms/step - accuracy: 1.0000 - loss: 0.0326 - val_accuracy: 1.0000 - val_loss: 0.0095
Epoch 4/11
445/445 ━━━━━━━━━━━━━━━━━━━━ 5s 10ms/step - accuracy: 1.0000 - loss: 0.0087 - val_accuracy: 1.0000 - val_loss: 0.0094
Epoch 5/11
445/445 ━━━━━━━━━━━━━━━━━━━━ 88s 196ms/step - accuracy: 1.0000 - loss: 0.0068 - val_accuracy: 1.0000 - val_loss: 0.0035
Epoch 6/11
445/445 ━━━━━━━━━━━━━━━━━━━━ 5s 12ms/step - accuracy: 1.0000 - loss: 0.0033 - val_accuracy: 1.0000 - val_loss: 0.0035
Epoch 7/11
445/445 ━━━━━━━━━━━━━━━━━━━━ 139s 203ms/step - accuracy: 1.0000 - loss: 0.0027 - val_accuracy: 1.0000 - val_loss: 0.0017
Epoch 8/11
445/445 ━━━━━━━━━━━━━━━━━━━━ 0s 344us/step - accuracy: 1.0000 - loss: 0.0016 - val_accuracy: 1.0000 - val_loss: 0.0017
Epoch 9/11
445/445 ━━━━━━━━━━━━━━━━━━━━ 134s 186ms/step - accuracy: 1.0000 - loss: 0.0014 - val_accuracy: 1.0000 - val_loss: 9.9276e-04
Epoch 10/11
445/445 ━━━━━━━━━━━━━━━━━━━━ 10s 23ms/step - accuracy: 1.0000 - loss: 9.2400e-04 - val_accuracy: 1.0000 - val_loss: 9.7778e-04
Epoch 11/11
445/445 ━━━━━━━━━━━━━━━━━━━━ 141s 207ms/step - accuracy: 1.0000 - loss: 8.0374e-04 - val_accuracy: 1.0000 - val_loss: 6.1276e-04
```

Line 1: Train the model

python
Copy code

```
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,  # Number of epochs
    callbacks=[early_stopping, model_checkpoint]  # Use callbacks
)
```

Explanation:
This line initiates the training process of the model using the `.fit()` method, which trains the model on data provided by the training generator and evaluates it on the validation data. The key arguments used in this code are:

- **`train_generator`**:
  This is the data generator for the training set, created earlier with `ImageDataGenerator.flow_from_directory`. It continuously provides batches of training images and their corresponding labels to the model.
- **`steps_per_epoch=train_generator.samples // train_generator.batch_size`**:
  Defines the number of steps (batches) the model will process in each epoch. It is calculated as the total number of training samples divided by the batch size. This determines how many batches of data are processed per epoch, ensuring the entire training dataset is seen by the model once each epoch.
- **`validation_data=valid_generator`**:
  Provides the validation data generator, which supplies batches of images and labels from the validation set. This data is used to evaluate the model's performance at the end of each epoch, giving insights into how well the model generalizes to unseen data.
- **`validation_steps=valid_generator.samples // valid_generator.batch_size`**:
  Specifies the number of validation steps per epoch, calculated as the total number of validation samples divided by the batch size. This ensures that the entire validation set is used during the validation phase of each epoch.
- **`epochs=11`**:
  Sets the number of epochs, which is the number of times the model will iterate over the entire training dataset. Here, it is set to 11, meaning the model will attempt to train over the full dataset up to 11 times unless stopped early by the `EarlyStopping` callback.
- **`callbacks=[early_stopping, model_checkpoint]`**:
  Adds the `EarlyStopping` and `ModelCheckpoint` callbacks to the training process:
  - **`EarlyStopping`**: Stops training early if the validation loss does not improve for a specified number of epochs (`patience`), preventing overfitting and saving time.
  - **`ModelCheckpoint`**: Saves the model with the best validation loss during training, ensuring that the best version of the model is preserved.

Code :

```
[ ]  # Evaluate the trained model on the test data
     test_loss, test_acc = model.evaluate(test_generator, steps=test_generator.samples // test_generator.batch_size)
     print(f"Test accuracy: {test_acc}")
```

Output :

```
24/24 ━━━━━━━━━━━━━━━━━━━━ 4s 152ms/step - accuracy: 1.0000 - loss: 6.1154e-04
Test accuracy: 1.0
```

Line 1: Evaluate the trained model on the test data

python
Copy code

```
test_loss, test_acc = model.evaluate(test_generator,
steps=test_generator.samples // test_generator.batch_size)
```

Explanation:
This line evaluates the performance of the trained model on the test dataset using the `.evaluate()` method, which computes the loss and metrics specified during model compilation. The key arguments are:

- **`test_generator`**:
  This is the data generator for the test set, created earlier using `ImageDataGenerator.flow_from_directory`. It provides batches of test images and their labels to the model for evaluation.
- **`steps=test_generator.samples // test_generator.batch_size`**:
  Specifies the number of steps (batches) that the evaluation process should run. It is calculated as the total number of test samples divided by the batch size. This ensures that the entire test dataset is used for evaluation.
- **`test_loss, test_acc`**:
  The method returns the evaluation results, with `test_loss` representing the loss value (binary crossentropy in this case) on the test set and `test_acc` representing the accuracy on the test data. These values provide a direct measure of how well the model performs on unseen test data.

Line 2: Print the test accuracy

python
Copy code

```
print(f"Test accuracy: {test_acc}")
```

Explanation:
This line prints the test accuracy obtained from the evaluation step, formatted as a floating-point number. Displaying the test accuracy helps assess how well the model generalizes to completely unseen data, which is a critical aspect of model validation.

Code :

```
from tensorflow.keras.applications import VGG19
from tensorflow.keras.preprocessing import image
from tensorflow.keras.applications.vgg19 import preprocess_input, decode_predictions
import numpy as np

# Load the VGG19 model pre-trained on ImageNet
model = VGG19(weights='imagenet')
```

Output :

```
Downloading data from https://storage.googleapis.com/tensorflow/keras-applications/vgg19/vgg19_weights_tf_dim_ordering_tf_kernels.h
574710816/574710816 ━━━━━━━━━━━━━━━━━━━━ 11s 0us/step
```

Line 1: Import the VGG19 model from Keras applications

python
Copy code

```
from tensorflow.keras.applications import VGG19
```

Explanation:
This line imports the `VGG19` model class from `tensorflow.keras.applications`. The `VGG19` is a popular deep convolutional neural network architecture known for its 19 layers, including 16 convolutional and 3 fully connected layers. It is widely used for image classification tasks due to its strong feature extraction capabilities.

Line 2: Import image preprocessing utilities

python
Copy code

```
from tensorflow.keras.preprocessing import image
```

Explanation:
This line imports the `image` module from `tensorflow.keras.preprocessing`, which provides utilities for loading, processing, and augmenting images. These utilities are useful when preparing images to feed into a model for prediction.

Line 3: Import specific functions for preprocessing and decoding predictions

python
Copy code
```
from tensorflow.keras.applications.vgg19 import preprocess_input, decode_predictions
```

Explanation:
This line imports two specific functions from `tensorflow.keras.applications.vgg19`:

- **`preprocess_input`**: Prepares an input image in the format expected by the VGG19 model (e.g., scaling pixel values, adjusting channels). This function preprocesses the image to match the data format that the model was trained on (ImageNet in this case).
- **`decode_predictions`**: Decodes the model's output predictions into human-readable class labels with corresponding probabilities. It converts the raw prediction scores into a list of predicted objects with their labels and confidence levels.

Line 4: Import NumPy

python
Copy code
```
import numpy as np
```

Explanation:
This line imports the `numpy` library, which is widely used for numerical computations in Python. In this context, `numpy` may be used to handle image data arrays and manipulate pixel data during preprocessing.

Line 5: Load the VGG19 model pre-trained on ImageNet

python
Copy code
```
model = VGG19(weights='imagenet')
```

Explanation:
This line loads the `VGG19` model with pre-trained weights on the ImageNet dataset. ImageNet is a large-scale visual database used for image classification and object detection tasks. By using weights trained on ImageNet, the model can recognize and classify images into thousands of common categories without additional training. The pre-trained model serves as a powerful feature extractor that can make predictions on new images based on its extensive training on a wide range of object categories.

Code :

```
[ ]  def preprocess_and_predict(img_path):
         # Load and preprocess the image
         img = image.load_img(img_path, target_size=(224, 224))
         img_array = image.img_to_array(img)
         img_array = np.expand_dims(img_array, axis=0)
         img_array = preprocess_input(img_array)

         # Make prediction
         predictions = model.predict(img_array)
         decoded_predictions = decode_predictions(predictions, top=3)[0]  # Get top 3 predictions

         return decoded_predictions
```

Line 1: Function Definition

python
Copy code

```
def preprocess_and_predict(img_path):
```

Explanation:
This line defines a function named `preprocess_and_predict` that takes one argument, `img_path`, which is the file path of the image you want to classify using the VGG19 model. The function will preprocess the image, make predictions using the model, and return the top predicted classes.

Line 2-3: Load and preprocess the image

python
Copy code

```
# Load and preprocess the image
img = image.load_img(img_path, target_size=(224, 224))
```

Explanation:
This line loads the image from the specified `img_path` and resizes it to `(224, 224)` pixels, which is the required input size for the VGG19 model. The `target_size` parameter ensures that the image matches the expected input shape of the model.

Line 4: Convert the image to a NumPy array

python
Copy code
```
img_array = image.img_to_array(img)
```

Explanation:
This line converts the loaded image into a NumPy array, allowing the pixel data to be manipulated and used as input for the model. The resulting array will have the shape `(224, 224, 3)`, representing the height, width, and three color channels (RGB) of the image.

Line 5: Expand the dimensions of the array

python
Copy code
```
img_array = np.expand_dims(img_array, axis=0)
```

Explanation:
This line adds an extra dimension to the image array, making its shape `(1, 224, 224, 3)`. The extra dimension represents the batch size, which is required because the model expects input data in batches. Even when predicting a single image, it needs to be in the shape of a batch of images.

Line 6: Preprocess the input array for the VGG19 model

python
Copy code
```
img_array = preprocess_input(img_array)
```

Explanation:
This line applies the `preprocess_input` function to the image array, which adjusts the pixel values according to the preprocessing rules used when training the VGG19 model on ImageNet. For VGG19, this typically involves scaling pixel values and adjusting them based on the mean pixel values used during the model's training, making the data compatible with the model.

Line 7-8: Make predictions using the model

python
Copy code
```
# Make prediction
predictions = model.predict(img_array)
```

Explanation:
This line uses the preprocessed image array as input to the VGG19 model to make predictions. The `predict` method outputs an array of prediction scores corresponding to each class in the ImageNet dataset, representing the model's confidence in each class.

Line 9: Decode the predictions into human-readable labels

python
Copy code

```
decoded_predictions = decode_predictions(predictions, top=3)[0]  # Get top 3 predictions
```

Explanation:
This line decodes the raw prediction scores into human-readable class labels and probabilities using the `decode_predictions` function. The `top=3` parameter specifies that the function should return the top 3 predictions with the highest confidence scores. The `[0]` index is used to access the list of predictions for the first (and only) image in the batch.

Line 10: Return the decoded predictions

python
Copy code

```
return decoded_predictions
```

Explanation:
This line returns the decoded predictions as a list of tuples, each containing the class label, the class description, and the probability score. These predictions help interpret what the model believes the image depicts.

Code :

```python
import os
import random
from PIL import Image
import matplotlib.pyplot as plt
from tensorflow.keras.applications.vgg19 import decode_predictions

# 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()))
```

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

# Preprocess and make predictions
decoded_predictions = preprocess_and_predict(img_path)

# Display the image with the top predictions
plt.figure(figsize=(10, 5))
```

```python
# Display the image
plt.subplot(1, 2, 1)
plt.imshow(img)
plt.axis('off')
plt.title(f'Image - Class: {random_class}')

# Display the top predictions
plt.subplot(1, 2, 2)
plt.barh([p[1] for p in decoded_predictions], [p[2] for p in decoded_predictions])
plt.xlabel('Confidence')
plt.title('Top 3 Predictions')

plt.show()
```

Output:

Line 1 : Imports

python
Copy code

```
import os
import random
from PIL import Image
import matplotlib.pyplot as plt
from tensorflow.keras.applications.vgg19 import decode_predictions
```

Explanation:

- `os`: Provides functions to interact with the operating system, such as file manipulation.
- `random`: Provides functions for generating random numbers, which is used here to select random files.
- `PIL.Image`: The Python Imaging Library (Pillow) for opening and manipulating images.
- `matplotlib.pyplot`: A plotting library for creating visualizations.
- `decode_predictions` from `tensorflow.keras.applications.vgg19`: A utility for decoding model predictions into human-readable class labels.

Line 2 : Define paths to the directories containing images:

python
Copy code
```
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
```

Explanation:

- These lines define the file paths to directories where images of different classes (Soft, Hard, No_ex) are stored.

Line 3 : Dictionary mapping class names to directories:

python
Copy code
```
class_dirs = {
    'Soft': soft_dir,
    'Hard': hard_dir,
    'No_ex': no_ex_dir
}
```

Explanation:

- A dictionary `class_dirs` maps each class name to its respective directory path.

Line 4 : Randomly select a class:

python
Copy code
```
random_class = random.choice(list(class_dirs.keys()))
```

Explanation:

- Randomly selects one of the class names from the `class_dirs` dictionary.

Line 5 : Get all image files from the selected class directory:

python
Copy code
```
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'))]
```

Explanation:

- Lists all files in the directory of the randomly selected class.
- Filters the list to include only files with extensions `.png`, `.jpg`, or `.jpeg`.

Line 6 : Randomly select an image from the class directory

python
Copy code
```
random_image_file = random.choice(image_files)
```

Explanation:

- Randomly selects one image file from the filtered list of images.

Line 7 : Load the selected image

python
Copy code
```
img_path = os.path.join(class_dirs[random_class], random_image_file)
img = Image.open(img_path)
```

Explanation:

- Constructs the full path to the selected image file.
- Opens the image using Pillow's `Image.open()` method.

Line 8 : Preprocess and make predictions

python
Copy code
```
decoded_predictions = preprocess_and_predict(img_path)
```

Explanation:

- Calls a function `preprocess_and_predict()` to preprocess the image and obtain predictions. (Note: This function is not defined in the provided code; you need to define it.)

Line 9 : Display the image with the top predictions

python
Copy code

```
plt.figure(figsize=(10, 5))
```

- **Explanation:**

  Creates a new figure for plotting with a specified size.

python
Copy code

```
# Display the image
plt.subplot(1, 2, 1)
plt.imshow(img)
plt.axis('off')
plt.title(f'Image - Class: {random_class}')
```

- **Explanation:**
- Adds a subplot to the figure to display the image.
- Shows the image using `plt.imshow()`.
- Hides the axis with `plt.axis('off')`.
- Sets the title of the subplot to indicate the class of the image.

python
Copy code

```
# Display the top predictions
plt.subplot(1, 2, 2)
plt.barh([p[1] for p in decoded_predictions], [p[2] for p in
decoded_predictions])
plt.xlabel('Confidence')
plt.title('Top 3 Predictions')
```

- **Explanation:**
- Adds another subplot to the figure to display a horizontal bar chart of the top predictions.
- Uses `plt.barh()` to create the bar chart, where `[p[1] for p in decoded_predictions]` is the list of class labels, and `[p[2] for p in decoded_predictions]` is the list of confidence scores.
- Sets the x-axis label to 'Confidence'.

- Sets the title of the subplot to 'Top 3 Predictions'.

python
Copy code

```
plt.show()
```

- **Explanation:**
- Displays the figure with the image and bar chart.

Code:

```
import os
import random
from PIL import Image
import matplotlib.pyplot as plt
import shutil
import numpy as np
from tensorflow.keras.applications import VGG19
from tensorflow.keras.preprocessing import image
from tensorflow.keras.applications.vgg19 import preprocess_input, decode_predictions

# 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
}
```

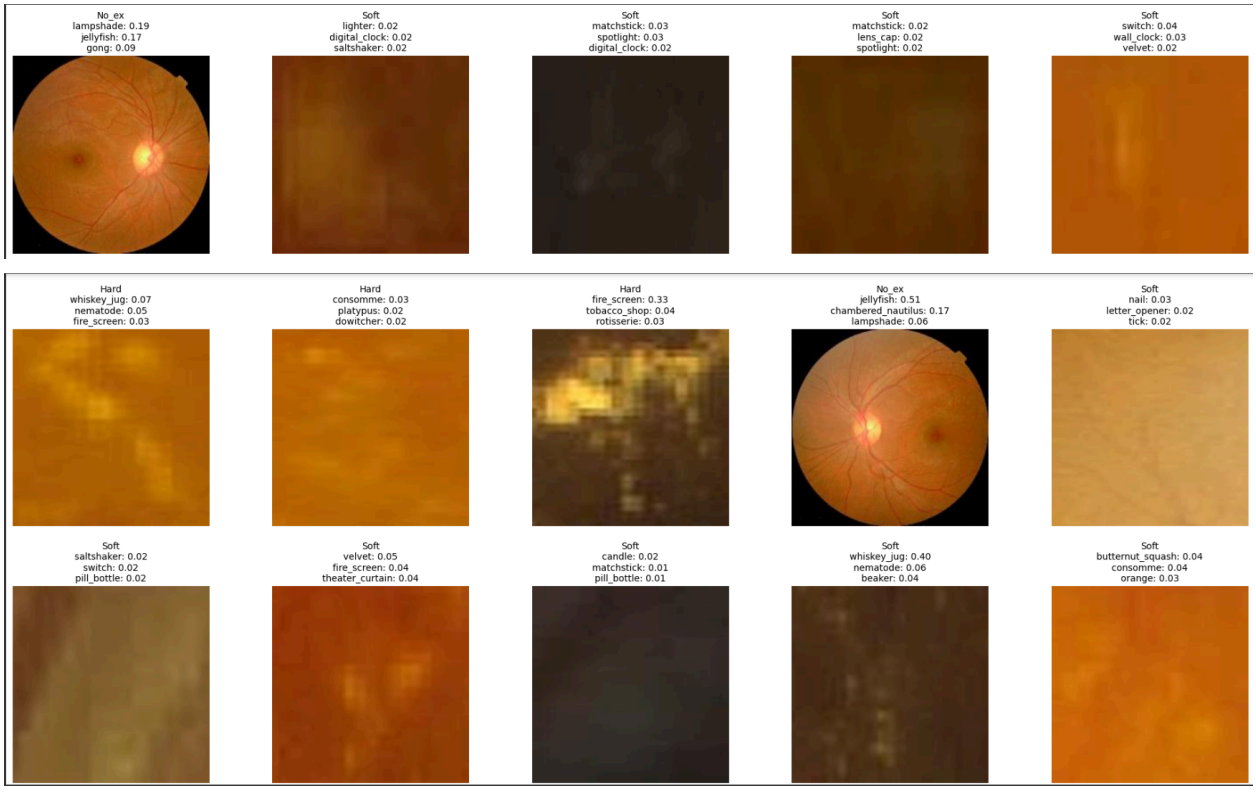
```
# Collect all images from the directories
all_images = []
for class_name, dir_path in class_dirs.items():
    image_files = os.listdir(dir_path)
    image_files = [f for f in image_files if f.endswith(('.png', '.jpg', '.jpeg'))]  # Filter image files
    all_images.extend([(class_name, os.path.join(dir_path,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

# Load the VGG19 model pre-trained on ImageNet
model = VGG19(weights='imagenet')

# Function to preprocess image and get predictions
def preprocess_and_predict(img_path):
    # Load and preprocess the image
    img = image.load_img(img_path, target_size=(224, 224))
    img_array = image.img_to_array(img)
    img_array = np.expand_dims(img_array, axis=0)
    img_array = preprocess_input(img_array)

    # Make prediction
    predictions = model.predict(img_array)
    decoded_predictions = decode_predictions(predictions, top=3)[0]  # Get top 3 predictions
```

```
    return decoded_predictions

# Copy selected images to the download directory and predict
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)

    # Get predictions for the image
    predictions = preprocess_and_predict(img_path)
    # Save the predictions for later use
    with open(os.path.join(download_dir, f"{os.path.basename(img_path)}_predictions.txt"), 'w') as f:
        for pred in predictions:
            f.write(f"{pred[1]}: {pred[2]:.2f}\n")

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

# Display the selected images with predictions in a clear format
plt.figure(figsize=(20, 12))
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')

    # Get predictions for the image
    predictions = preprocess_and_predict(img_path)
    pred_text = "\n".join([f"{pred[1]}: {pred[2]:.2f}" for pred in predictions])
```

```
    # Display class name and predictions with clear formatting
    plt.title(f'{class_name}\n{pred_text}', fontsize=10, ha='center')

plt.tight_layout(pad=2.0)
plt.show()
```

Output:

```
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 456ms/step
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 18ms/step
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 17ms/step
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 15ms/step
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 15ms/step
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 16ms/step
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 15ms/step
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 15ms/step
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 15ms/step
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 16ms/step
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 18ms/step
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 16ms/step
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 15ms/step
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 16ms/step
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 17ms/step
15 random images have been downloaded to /content/downloaded_images
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 20ms/step
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 18ms/step
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 17ms/step
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 17ms/step
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 16ms/step
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 17ms/step
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 15ms/step
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 17ms/step
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 16ms/step
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 17ms/step
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 16ms/step
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 21ms/step
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 16ms/step
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 17ms/step
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 17ms/step
```



Line 1:Import OS Module

python
Copy code
```
import os
```

- **Explanation:**

  Imports the `os` module, which provides a way to interact with the operating system, including

  file and directory operations.

Line 2:Import Random Module

python
Copy code
```
import random
```

- **Explanation:**

  Imports the `random` module, which includes functions to generate random numbers and make random selections.

Line 3:Import Image Class from PIL

python
Copy code
```
from PIL import Image
```

- **Explanation:**

  Imports the `Image` class from the `PIL` (Python Imaging Library) module, which is used to open, manipulate, and save image files.

Line 4:Import Pyplot from Matplotlib

python
Copy code
```
import matplotlib.pyplot as plt
```

- **Explanation:**
- Imports `matplotlib.pyplot` as `plt`, a plotting library used to create visualizations such as graphs and charts.

Line 5: Import Shutil Module

python
Copy code
```
import shutil
```

- **Explanation**:
- Imports the `shutil` module, which provides high-level file operations such as copying and removing files.

Line 6: Import NumPy Library

Python

Copy code

```
import numpy as np
```

- **Explanation**:
- Imports the `numpy` library as `np`, which is used for numerical operations on arrays.
- Sure! Let's break down the code line by line with explanations:

Line 7: Import VGG19 Model from Keras Applications

python
Copy code
```
from tensorflow.keras.applications import VGG19
```

- **Explanation**:
- Imports the VGG19 model from TensorFlow's Keras applications, a pre-trained deep learning model for image classification.

Line 8: Import Image Utilities from Keras

python
Copy code
```
from tensorflow.keras.preprocessing import image
```

- **Explanation:**
- Imports the `image` module from Keras preprocessing, which includes utilities for loading and processing images.

Line 9:Import Preprocessing and Prediction Utilities for VGG19

python
Copy code

```
from tensorflow.keras.applications.vgg19 import preprocess_input,
decode_predictions
```

- **Explanation:**
- Imports `preprocess_input` and `decode_predictions` functions specific to the VGG19 model for image preprocessing and interpreting model predictions.

Line 10:Define Directory Paths for Image Classes,

python
Copy code

```
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
```

- **Explanation:**
- Define paths to directories containing images for different classes: 'Soft', 'Hard', and 'No_ex'.

Line 11. Create Dictionary Mapping Class Names to Directories

python
Copy code

```
class_dirs = {

    'Soft': soft_dir,

    'Hard': hard_dir,
```

```
    'No_ex': no_ex_dir
}
```

- **Explanation:**
- Creates a dictionary `class_dirs` that maps class names to their corresponding directory paths.

Line 12. Collect All Images from Directories

python
Copy code

```
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])
```

- **Explanation:**
- Iterates over each class directory to collect all image files. Filters images by extension and stores their paths and class names in `all_images`.

Line 13. Randomly Select 15 Images

python
Copy code

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

- **Explanation:**
- Randomly selects up to 15 images from the `all_images` list. Ensures no error if fewer than 15 images are available.

Line 14. Define Directory Path for Downloaded Images

python
Copy code

```
download_dir = '/content/downloaded_images'  # Change path if needed
```

- **Explanation:**
- Defines the path where selected images will be saved. Adjust this path as needed.

Line 15. Create Directory for Downloaded Images

python
Copy code

```
os.makedirs(download_dir, exist_ok=True)
```

- **Explanation:**
- Creates the `download_dir` directory if it does not exist. The `exist_ok=True` parameter prevents errors if the directory already exists.

16. Load Pre-trained VGG19 Model

python
Copy code

```
model = VGG19(weights='imagenet')
```

- **Explanation:**
- Initializes the VGG19 model with weights pre-trained on the ImageNet dataset, enabling image classification.

17. Define Function to Preprocess Image and Make Predictions

python
Copy code

```
def preprocess_and_predict(img_path):

    # Load and preprocess the image

    img = image.load_img(img_path, target_size=(224, 224))
```

```
    img_array = image.img_to_array(img)

    img_array = np.expand_dims(img_array, axis=0)

    img_array = preprocess_input(img_array)

    # Make prediction

    predictions = model.predict(img_array)

    decoded_predictions = decode_predictions(predictions, top=3)[0]  # Get top 3 predictions

    return decoded_predictions
```

- **Explanation:**
- Defines a function that:
- Loads and resizes an image to 224x224 pixels.
- Converts the image to an array and preprocesses it for the VGG19 model.
- Makes predictions and decodes the top 3 predictions.

18. Copy Selected Images and Save Predictions

python
Copy code

```
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)

    # Get predictions for the image

    predictions = preprocess_and_predict(img_path)

    # Save the predictions for later use
```

```
    with open(os.path.join(download_dir,
f"{os.path.basename(img_path)}_predictions.txt"), 'w') as f:

        for pred in predictions:

            f.write(f"{pred[1]}: {pred[2]:.2f}\n")
```

Explanation: Iterates over the randomly selected images:

1. Copies each image to `download_dir` with a unique filename.
2. Retrieves predictions for each image and saves them to a text file in `download_dir`.

19. Print Confirmation Message

python
Copy code

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

- **Explanation:**
- Prints a message confirming that 15 random images have been downloaded to the specified directory.

20. Display Images with Predictions

python
Copy code

```
plt.figure(figsize=(20, 12))

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')

    # Get predictions for the image

    predictions = preprocess_and_predict(img_path)

    pred_text = "\n".join([f"{pred[1]}: {pred[2]:.2f}" for pred in
predictions])

    # Display class name and predictions with clear formatting

    plt.title(f'{class_name}\n{pred_text}', fontsize=10, ha='center')

plt.tight_layout(pad=2.0)

plt.show()
```

- **Explanation:**
- Creates a figure to display images in a 3x5 grid. For each image:
- Loads and displays the image.
- Retrieves and formats predictions.
- Sets the title to include the class name and predictions.
- Adjusts layout and shows the plot.

List of issues

List of issues faced during the development process how they are tackled whether they are resolved or not .

1.Memory or Resource Limitations:

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

Resolution: Optimize the dataset.

2.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.

Resolution: Check the dataset size and adjust the selection logic as necessary. Ensure that each class has a sufficient number of images.

3. Issue: Image Processing and Model Input Mismatch

Issue : The image size or format might not match the expected input of the model. Solution: Resize images to the expected dimensions (224x224 for VGG19) and ensure correct preprocessing. Verify that the images are in the required format (JPEG, PNG).

Resolution: Ensure images are resized and preprocessed correctly before feeding them into the model.

4. Issue: Model Training and Overfitting

Issue : The model might overfit or underfit due to the number of layers or insufficient data. Solution: Use techniques like dropout, early stopping, and data augmentation to mitigate overfitting. Adjust the model architecture if needed.

Resolution: Overfitting can be managed with regularization techniques, and underfitting might require a more complex model or additional data.

5. Issue: Model Saving and Loading Errors

Problem: Issues might occur when saving or loading models. Solution: Use proper file paths and ensure the model saving/loading functions are correctly used. Handle exceptions that might occur during these operations.

Resolution: Confirm that model files are saved correctly and can be loaded without errors.

6. Issue: Plotting Errors

Problem: Issues might arise when plotting images or predictions.

Solution: Ensure that image files exist and are correctly opened. Verify that matplotlib functions are used correctly and that there are no mismatches in data types.

Resolution: Debug plotting issues by checking image paths and ensuring data passed to plotting functions are valid.