

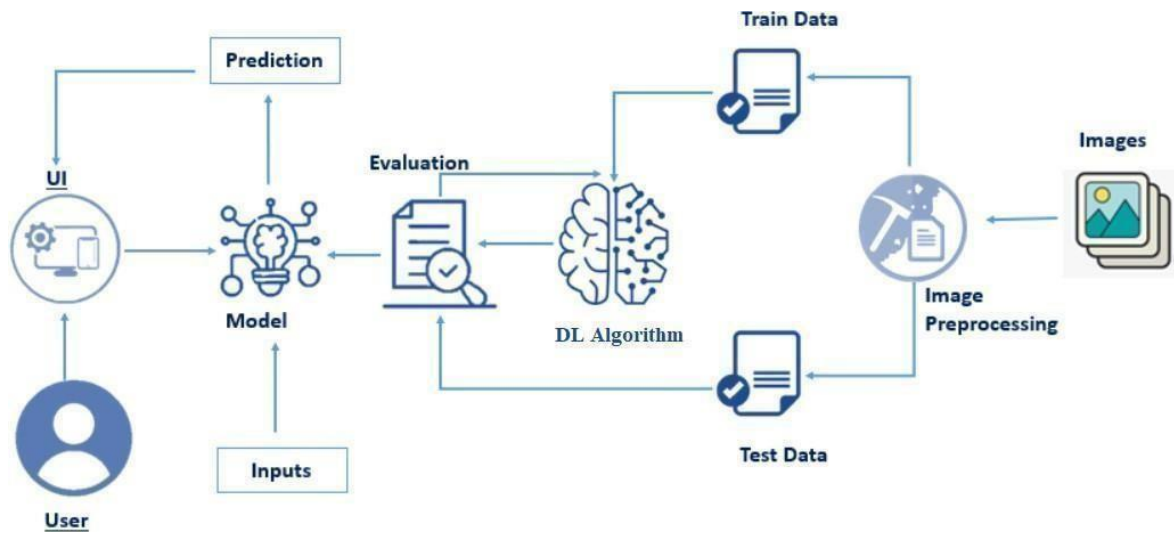
Eye Disease Detection Using Deep Learning

Project Description:

In this project we are classifying various types of Eye Diseases that people get due to various reasons like age, diabetes, etc. These diseases are majorly classified into 4 categories namely Normal, cataract, Diabetic Retinopathy & Glaucoma. Deep-learning (DL) methods in artificial intelligence (AI) play a dominant role as high-performance classifiers in the detection of the Eye Diseases using images.

Transfer learning has become one of the most common techniques that has achieved better performance in many areas, especially in image analysis and classification. We used Transfer Learning techniques like Inception V3, VGG19, Xception V3 that are more widely used as a transfer learning method in image analysis and they are highly effective.

Technical Architecture:



Project Flow:

- The user interacts with the UI (User Interface) to choose the image.
- The chosen image analyzed by the model which is integrated with flask application.
- The VGG19 Model analyzes the image, then the prediction is showcased on the Flask UI.

To accomplish this, we have to complete all the activities and tasks listed below

- Data Collection.
 - Create a Train and Test path.
- Image Pre-processing.
 - Import the required library
 - Configure ImageDataGenerator class
 - Apply ImageDataGenerator functionality to Trainset and Testset
- Model Building
 - Pre-trained CNN model as a Feature Extractor
 - Adding Dense Layer
 - Configure the Learning Process
 - Train the model
 - Save the Model
 - Test the model
- Application Building
 - Create an HTML file

Prior Knowledge:

You must have prior knowledge of following topics to complete this project.

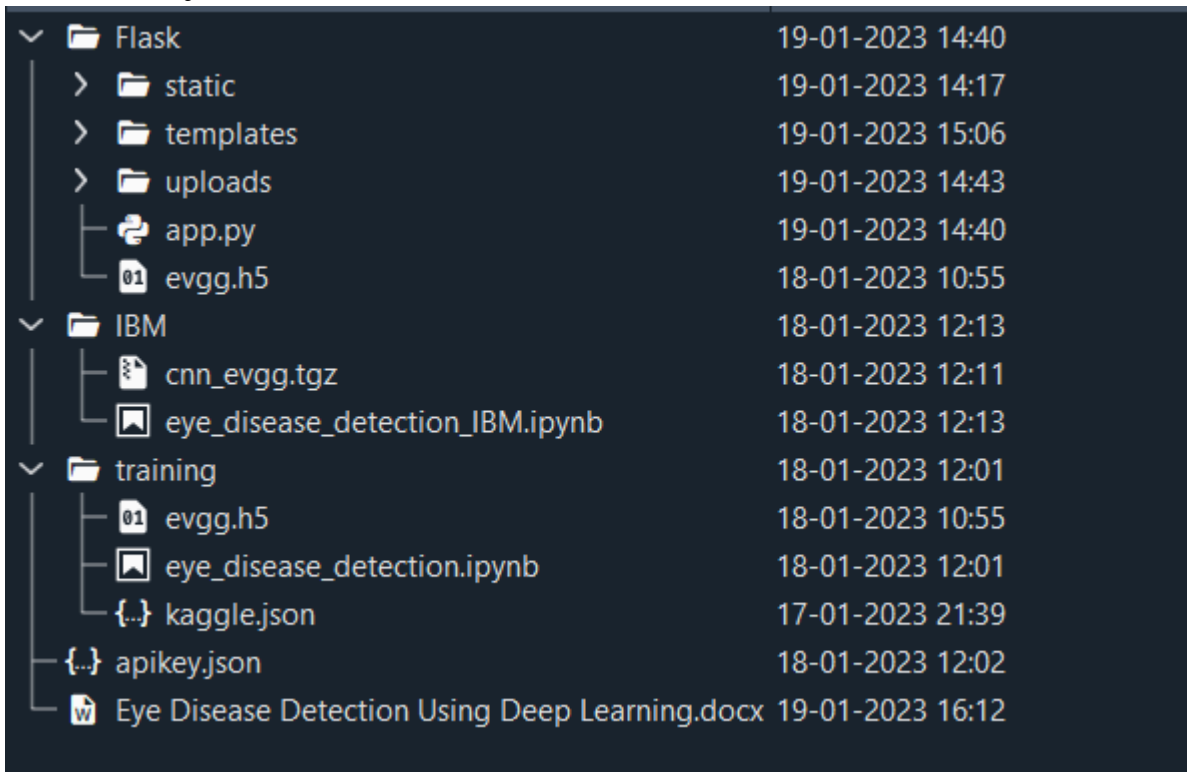
- **Deep Learning Concepts**
 - **CNN:** <https://towardsdatascience.com/basics-of-the-classic-cnn-a3dce1225add>
 - **VGG19:** [VGG-19 convolutional neural network - MATLAB vgg19 - MathWorks India](#)
 - **ResNet-50V2:** <https://towardsdatascience.com/understanding-and-coding-a-resnet-in-keras-446d7ff84d33>
 - **Inception-V3:** <https://iq.opengenus.org/inception-v3-model-architecture/>
 - **Xception:** <https://pyimagesearch.com/2017/03/20/imagenet-vggnet-resnet-inception-xception-keras/>
- **Flask:** Flask is a popular Python web framework, meaning it is a third-party Python library used for developing web applications.

Link: https://www.youtube.com/watch?v=lj4I_CvBnt0

Build Python Code

Project Structure:

Create a Project folder which contains files as shown below



The screenshot shows a file explorer interface with a dark background. It displays a hierarchical project structure. The 'Flask' folder is expanded, showing subfolders 'static' and 'templates', and files 'app.py' and 'evgg.h5'. The 'IBM' folder is also expanded, showing 'cnn_evgg.tgz' and 'eye_disease_detection_IBM.ipynb'. The 'training' folder is expanded, showing 'evgg.h5', 'eye_disease_detection.ipynb', and 'kaggle.json'. At the root level, there are files 'apikey.json' and 'Eye Disease Detection Using Deep Learning.docx'. Each item is accompanied by a timestamp.

Flask	19-01-2023 14:40
static	19-01-2023 14:17
templates	19-01-2023 15:06
uploads	19-01-2023 14:43
app.py	19-01-2023 14:40
evgg.h5	18-01-2023 10:55
IBM	18-01-2023 12:13
cnn_evgg.tgz	18-01-2023 12:11
eye_disease_detection_IBM.ipynb	18-01-2023 12:13
training	18-01-2023 12:01
evgg.h5	18-01-2023 10:55
eye_disease_detection.ipynb	18-01-2023 12:01
kaggle.json	17-01-2023 21:39
apikey.json	18-01-2023 12:02
Eye Disease Detection Using Deep Learning.docx	19-01-2023 16:12

- The Dataset folder contains the training and testing images for training our model.
- For building a Flask Application we need HTML pages stored in the **templates** folder, CSS for styling the pages stored in the static folder and a python script **app.py** for server side scripting
- The IBM folder consists of a trained model notebook on IBM Cloud.
- Training folder consists of eye_disease_detection.ipynb model training file & evgg.h5 is saved model

Milestone 1: Data Collection

There are many popular open sources for collecting the data. Eg: kaggle.com, UCI repository, etc.

Activity 1: Download the dataset

Collect images of Eye Diseases then organize into subdirectories based on their respective names as shown in the project structure. Create folders of types of Eye Diseases that need to be recognized.

In this project, we have collected images of 4 types of Eye Diseases images like Normal, cataract, Diabetic Retinopathy & Glaucoma and they are saved in the respective sub directories with their respective names.

You can download the dataset used in this project using the below link

Dataset:- <https://www.kaggle.com/datasets/gunavenkatdoddi/eye-diseases-classification>

Note: For better accuracy train on more images

We are going to build our training model on Google colab.

We will be connecting Kaggle with Google Colab because the dataset is too big to import using the following code:

```
[1]: !pip install tensorflow
```

```
[2]: !pip install matplotlib
```

```
Requirement already satisfied: matplotlib in c:\users\prajw\appdata\local\programs
```

Activity 2: Create training and testing dataset

To build a DL model we have to split training and testing data into two separate folders. But in this project dataset folder training and testing folders are not present. So, in this case we have to separate the data into train & test folders.

```
import splitfolders

splitfolders.ratio('/content/dataset', output="output", seed=1337, ratio=(.8, 0.2))

Copying files: 4217 files [00:02, 1558.61 files/s]
```

Four different transfer learning models are used in our project and the best model (VGG19) is selected. The image input size of VGG19 model is 224, 224.

```
IMAGE_SIZE = [224, 224]
```

Milestone 2: Image Preprocessing

In this milestone we will be improving the image data that suppresses unwilling distortions or enhances some image features important for further processing, although perform some geometric transformations of images like rotation, scaling, translation, etc.

Activity 1: Importing the libraries

Import the necessary libraries as shown in the image

```
import splitfolders
from tensorflow.keras.preprocessing.image import ImageDataGenerator
from PIL import ImageFile
ImageFile.LOAD_TRUNCATED_IMAGES = True
from tensorflow.keras.applications.vgg19 import VGG19, preprocess_input
from tensorflow.keras.layers import Flatten, Dense
from tensorflow.keras.models import Model
from tensorflow.keras.models import load_model
from tensorflow.keras.preprocessing import image
import numpy as np
import matplotlib.pyplot as plt
import matplotlib.image as mpimg
%matplotlib inline
```

Activity 2: Configure ImageDataGenerator class

ImageDataGenerator class is instantiated and the configuration for the types of data augmentation

There are five main types of data augmentation techniques for image data; specifically:

- Image shifts via the width_shift_range and height_shift_range arguments.
- The image flips via the horizontal_flip and vertical_flip arguments.
- Image rotations via the rotation_range argument
- Image brightness via the brightness_range argument.
- Image zoom via the zoom_range argument.

An instance of the ImageDataGenerator class can be constructed for train and test.

```
from tensorflow.keras.preprocessing.image import ImageDataGenerator
train_datagen = ImageDataGenerator(rescale = 1./255,
                                   shear_range = 0.2, zoom_range = 0.2,
                                   horizontal_flip = True)

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

Activity 3: Apply ImageDataGenerator functionality to Train set and Test set

Let us apply ImageDataGenerator functionality to the Train set and Test set by using the following code. For Training set using flow_from_directory function.

This function will return batches of images from the subdirectories

Arguments:

- directory: Directory where the data is located. If labels are "inferred", it should contain subdirectories, each containing images for a class. Otherwise, the directory structure is ignored.
- batch_size: Size of the batches of data which is 64.
- target_size: Size to resize images after they are read from disk.
- class_mode:
 - 'int': means that the labels are encoded as integers (e.g. for sparse_categorical_crossentropy loss).
 - 'categorical' means that the labels are encoded as a categorical vector (e.g. for categorical_crossentropy loss).
 - 'binary' means that the labels (there can be only 2) are encoded as float32 scalars with values 0 or 1 (e.g. for binary_crossentropy).
 - None (no labels).

```
training_set = train_datagen.flow_from_directory(  
    '/content/output/train',  
                                            target_size = (224, 224),  
                                            batch_size = 64,  
                                            class_mode = 'categorical')  
  
test_set = test_datagen.flow_from_directory('/content/output/val',  
                                            target_size = (224, 224),  
                                            batch_size = 64,  
                                            class_mode = 'categorical')  
  
Found 3372 images belonging to 4 classes.  
Found 845 images belonging to 4 classes.
```

Total the dataset is having 3372 train images, 845 test images divided under 4 classes.

Milestone 3: Model Building

Now it's time to build our model. Let's use the pre-trained model which is VGG19, one of the convolution neural net (CNN) architecture which is considered as a very good model for Image classification.

Deep understanding on the VGG19 model – Link is referred to in the prior knowledge section. Kindly refer to it before starting the model building part.

Activity 1: Pre-trained CNN model as a Feature Extractor

For one of the models, we will use it as a simple feature extractor by freezing all the five convolution blocks to make sure their weights don't get updated after each epoch as we train our own model.

Here, we have considered images of dimension (224,224,3).

Also, we have assigned `include_top = False` because we are using convolution layer for features extraction and wants to train fully connected layer for our image classification (since it is not the part of Imagenet dataset)

Flatten layer flattens the input. Does not affect the batch size.

```
VGG19 = VGG19(input_shape=IMAGE_SIZE + [3], weights='imagenet',include_top=False)

Downloading data from https://storage.googleapis.com/tensorflow/keras-applications/vgg19/vgg19\_80134624/80134624 [=====] - 1s 0us/step

for layer in VGG19.layers:
    layer.trainable = False

x = Flatten()(VGG19.output)
```

Activity 2: Adding Dense Layers

```
x = Flatten()(VGG19.output)

prediction = Dense(4, activation='softmax')(x)

model = Model(inputs=VGG19.input, outputs=prediction)
```

A dense layer is a deeply connected neural network layer. It is the most common and frequently used layer. Let us create a model object named model with inputs as VGG19.input and output as dense layer.

The number of neurons in the Dense layer is the same as the number of classes in the training set. The neurons in the last Dense layer, use softmax activation to convert their outputs into respective probabilities. Understanding the model is a very important phase to properly use it for training and prediction purposes.

Keras provides a simple method, summary to get the full information about the model and its layers.

Model: "sequential_2"

Layer (type)	Output Shape	Param #
=====		
sequential (Sequential)	(32, 256, 256, 3)	0
conv2d (Conv2D)	(32, 254, 254, 32)	896
max_pooling2d (MaxPooling2D)	(32, 127, 127, 32)	0
conv2d_1 (Conv2D)	(32, 125, 125, 64)	18496
max_pooling2d_1 (MaxPooling2D)	(32, 62, 62, 64)	0
conv2d_2 (Conv2D)	(32, 60, 60, 64)	36928
max_pooling2d_2 (MaxPooling2D)	(32, 30, 30, 64)	0
conv2d_3 (Conv2D)	(32, 28, 28, 64)	36928
max_pooling2d_3 (MaxPooling2D)	(32, 14, 14, 64)	0
conv2d_4 (Conv2D)	(32, 12, 12, 64)	36928
max_pooling2d_4 (MaxPooling2D)	(32, 6, 6, 64)	0
conv2d_5 (Conv2D)	(32, 4, 4, 64)	36928
max_pooling2d_5 (MaxPooling2D)	(32, 2, 2, 64)	0
flatten (Flatten)	(32, 256)	0
dense (Dense)	(32, 64)	16448
dense_1 (Dense)	(32, 4)	260

=====

Total params: 183812 (718.02 KB)

Trainable params: 183812 (718.02 KB)

Activity 3: Configure the Learning Process

The compilation is the final step in creating a model. Once the compilation is done, we can move on to the training phase. The loss function is used to find errors or deviations in the learning process. Keras requires a loss function during the model compilation process.

Optimization is an important process that optimizes the input weights by comparing the prediction and the loss function. Here we are using adam optimizer

Metrics are used to evaluate the performance of your model. It is similar to the loss function, but not used in the training process

```
27]: model.compile(  
    optimizer='adam',  
    loss=tf.keras.losses.SparseCategoricalCrossentropy(from_logits=False),  
    metrics=['accuracy']  
)
```

```
WARNING:tensorflow:From C:\Users\prajw\AppData\Local\Programs\Python\Python310\lib\site-packages\keras\src\optimizers\_init_.py:309:  
n.Optimizer is deprecated. Please use tf.compat.v1.train.Optimizer instead.
```

Activity 4: Train the model

Now, let us train our model with our image dataset. The model is trained for 50 epochs and after every epoch, the current model state is saved if the model has the least loss encountered till that time. We can see that the training loss decreases in almost every epoch and probably there is further scope to improve the model.

.fit functions used to train a deep learning neural network

Arguments:

- **steps_per_epoch**: it specifies the total number of steps taken from the generator as soon as one epoch is finished and the next epoch has started. We can calculate the value of **steps_per_epoch** as the total number of samples in your dataset divided by the batch size.
- **Epochs**: an integer and number of epochs we want to train our model for.
- **validation_data** can be either:
 - an inputs and targets list
 - a generator
 - an inputs, targets, and **sample_weights** list which can be used to evaluate the loss and metrics for any model after any epoch has ended.
- **validation_steps**: only if the **validation_data** is a generator then only this argument can be used. It specifies the total number of steps taken from the generator before it is stopped at every epoch and its value is calculated as the total number of validation data points in your dataset divided by the validation batch size.

```

history = model.fit(
    train_ds,
    batch_size=BATCH_SIZE,
    validation_data=val_ds,
    verbose=1,
    epochs=50,
)

```

```

105/105 [=====] - 180s 1s/step - loss: 1.3044 - accuracy: 0.3656 - val_loss: 1.0814 - val_accuracy: 0.4928
Epoch 2/50
105/105 [=====] - 100s 952ms/step - loss: 1.0176 - accuracy: 0.5470 - val_loss: 0.8324 - val_accuracy: 0.6490
Epoch 3/50
105/105 [=====] - 101s 963ms/step - loss: 0.8456 - accuracy: 0.6251 - val_loss: 0.7303 - val_accuracy: 0.7043
Epoch 4/50
105/105 [=====] - 100s 952ms/step - loss: 0.7263 - accuracy: 0.6904 - val_loss: 0.6186 - val_accuracy: 0.7404
Epoch 5/50
105/105 [=====] - 100s 953ms/step - loss: 0.6923 - accuracy: 0.7077 - val_loss: 0.5417 - val_accuracy: 0.7692
Epoch 6/50
105/105 [=====] - 100s 950ms/step - loss: 0.6299 - accuracy: 0.7381 - val_loss: 0.5675 - val_accuracy: 0.7620
Epoch 7/50
105/105 [=====] - 100s 947ms/step - loss: 0.6109 - accuracy: 0.7522 - val_loss: 0.5667 - val_accuracy: 0.7716
Epoch 8/50
105/105 [=====] - 100s 954ms/step - loss: 0.5699 - accuracy: 0.7650 - val_loss: 0.4661 - val_accuracy: 0.8149
Epoch 9/50
105/105 [=====] - 100s 952ms/step - loss: 0.5346 - accuracy: 0.7763 - val_loss: 0.4545 - val_accuracy: 0.8077
Epoch 10/50
105/105 [=====] - 100s 948ms/step - loss: 0.4942 - accuracy: 0.7969 - val_loss: 0.4506 - val_accuracy: 0.8269
Epoch 11/50
105/105 [=====] - 100s 951ms/step - loss: 0.4905 - accuracy: 0.8055 - val_loss: 0.3766 - val_accuracy: 0.8582
Epoch 12/50
105/105 [=====] - 100s 953ms/step - loss: 0.4660 - accuracy: 0.8109 - val_loss: 0.4252 - val_accuracy: 0.8558
Epoch 13/50
105/105 [=====] - 100s 951ms/step - loss: 0.4639 - accuracy: 0.8130 - val_loss: 0.3706 - val_accuracy: 0.8606
Epoch 14/50
105/105 [=====] - 99s 944ms/step - loss: 0.4574 - accuracy: 0.8145 - val_loss: 0.4249 - val_accuracy: 0.8317
Epoch 15/50
105/105 [=====] - 99s 943ms/step - loss: 0.4368 - accuracy: 0.8196 - val_loss: 0.3497 - val_accuracy: 0.8582
Epoch 16/50
105/105 [=====] - 100s 949ms/step - loss: 0.4257 - accuracy: 0.8270 - val_loss: 0.4173 - val_accuracy: 0.8341
Epoch 17/50
105/105 [=====] - 100s 946ms/step - loss: 0.4233 - accuracy: 0.8294 - val_loss: 0.3416 - val_accuracy: 0.8726
Epoch 18/50
105/105 [=====] - 101s 961ms/step - loss: 0.4081 - accuracy: 0.8294 - val_loss: 0.3763 - val_accuracy: 0.8462
Epoch 19/50
105/105 [=====] - 100s 950ms/step - loss: 0.4073 - accuracy: 0.8312 - val_loss: 0.3732 - val_accuracy: 0.8654
Epoch 20/50
105/105 [=====] - 100s 947ms/step - loss: 0.4101 - accuracy: 0.8339 - val_loss: 0.3792 - val_accuracy: 0.8438
Epoch 21/50
105/105 [=====] - 100s 951ms/step - loss: 0.4074 - accuracy: 0.8366 - val_loss: 0.3888 - val_accuracy: 0.8534
Epoch 22/50
105/105 [=====] - 99s 944ms/step - loss: 0.4097 - accuracy: 0.8312 - val_loss: 0.3439 - val_accuracy: 0.8534
Epoch 23/50
105/105 [=====] - 100s 949ms/step - loss: 0.3954 - accuracy: 0.8410 - val_loss: 0.3794 - val_accuracy: 0.8654

```

Activity 5: Save the Model

Out of all the models we tried (CNN, VGG19, Resnet50 V2, Inception V3 & Xception) VGG19 gave us the best accuracy.

```
model.save('evgg.h5')
```

So we are saving VGG19 as our final model

The model is saved with .h5 extension as follows

An H5 file is a data file saved in the Hierarchical Data Format (HDF). It contains multidimensional arrays of scientific data.

Testing the model:

Evaluation is a process during the development of the model to check whether the model is the best fit for the given problem and corresponding data. Load the saved model using load_model.

```
from tensorflow.keras.models import load_model
from tensorflow.keras.preprocessing import image
import numpy as np
model = load_model("/content/evgg.h5")
```

Taking an image as input and checking the results

```
img = image.load_img(r"/content/output/val/normal/2365_left.jpg",target_size= (224,224))#loading of the image
x = image.img_to_array(img)#image to array
x = np.expand_dims(x,axis = 0)#changing the shape
preds=model.predict(x)
pred=np.argmax(preds,axis=1)
index=['cataract','diabetic_retinopathy','glaucoma','normal']
result=str(index[pred[0]])
result

1/1 [=====] - 1s 806ms/step
'normal'
```

So our model has predicted the label correctly as Nomal.

Milestone 4: Application Building

In this section, we will be building a web application that is integrated to the model we built. A UI is provided for the uses where he has to enter the values for predictions. The enter values are given to the saved model and prediction is showcased on the UI.

This section has the following tasks

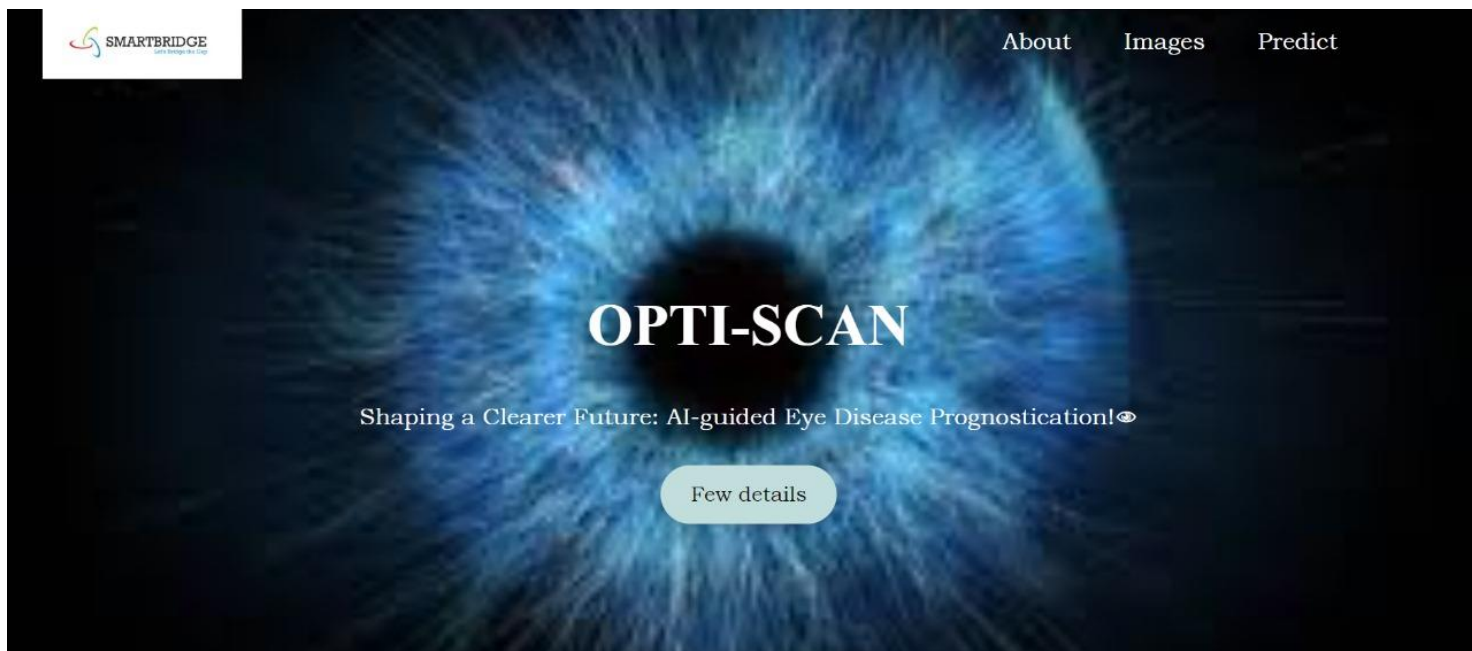
- Building HTML Pages
- Building python code
- Run the programme

Activity1: Building HTML Pages:

For this project create one HTML file namely

- index.html

Let's see how our index.html page looks like:



Eye 🧐

Early ocular disease detection is an economic and effective way to prevent blindness caused by diabetes, glaucoma, cataract, age-related macular degeneration(AMD) and many other diseases.

Types of Eyes

Images 📷



Activity 2: Build Python code:

Import the libraries

```
import numpy as np
import os
from flask import Flask, request, render_template
from tensorflow.keras.models import load_model
from tensorflow.keras.preprocessing import image
from tensorflow.keras.applications.resnet_v2 import preprocess_input
```

Loading the saved model and initializing the flask app

RenderML pages:

```
MODEL = tf.keras.models.load_model("../saved_models/1")
CLASS_NAMES = ["normal", "glaucoma", "diabetic_retinopathy", "cataract"]
endpoint = "your_endpoint_url"
```

```

@app.route('/')
def index():
    return render_template('index.html')

@app.route('/home')
def home():
    return render_template("index.html")

@app.route('/inp')
def inp():
    return render_template("img_input.html")

```

Once we uploaded the file into the app, then verifying the file uploaded properly or not. Here we will be using declared constructor to route to the HTML page which we have created earlier.

In the above example, '/' URL is bound with index.html function. Hence, when the home page of the web server is opened in browser, the html page will be rendered. Whenever you enter the values from the html page the values can be retrieved using POST Method.

```

@app.route('/predict', methods=["GET", "POST"])
def res():
    if request.method=="POST":
        f=request.files['image']
        basepath=os.path.dirname(__file__)
        filepath=os.path.join(basepath,'uploads',f.filename)
        f.save(filepath)

        img=image.load_img(filepath,target_size=(224,224,3))
        x=image.img_to_array(img)
        x=np.expand_dims(x,axis=0)

        img_data=preprocess_input(x)
        prediction=np.argmax(model.predict(img_data), axis=1)

        index=['cataract','diabetic_retinopathy','glaucoma','normal']

        result=str(index[prediction[0]])
        print(result)
        return render_template('output.html', prediction=result)

```

Here we are routing our app to predict function. This function retrieves all the values from the HTML page using Post request. That is stored in an array. This array is passed to the model.predict() function. This function returns the prediction. And this prediction value will rendered to the text that we have

:

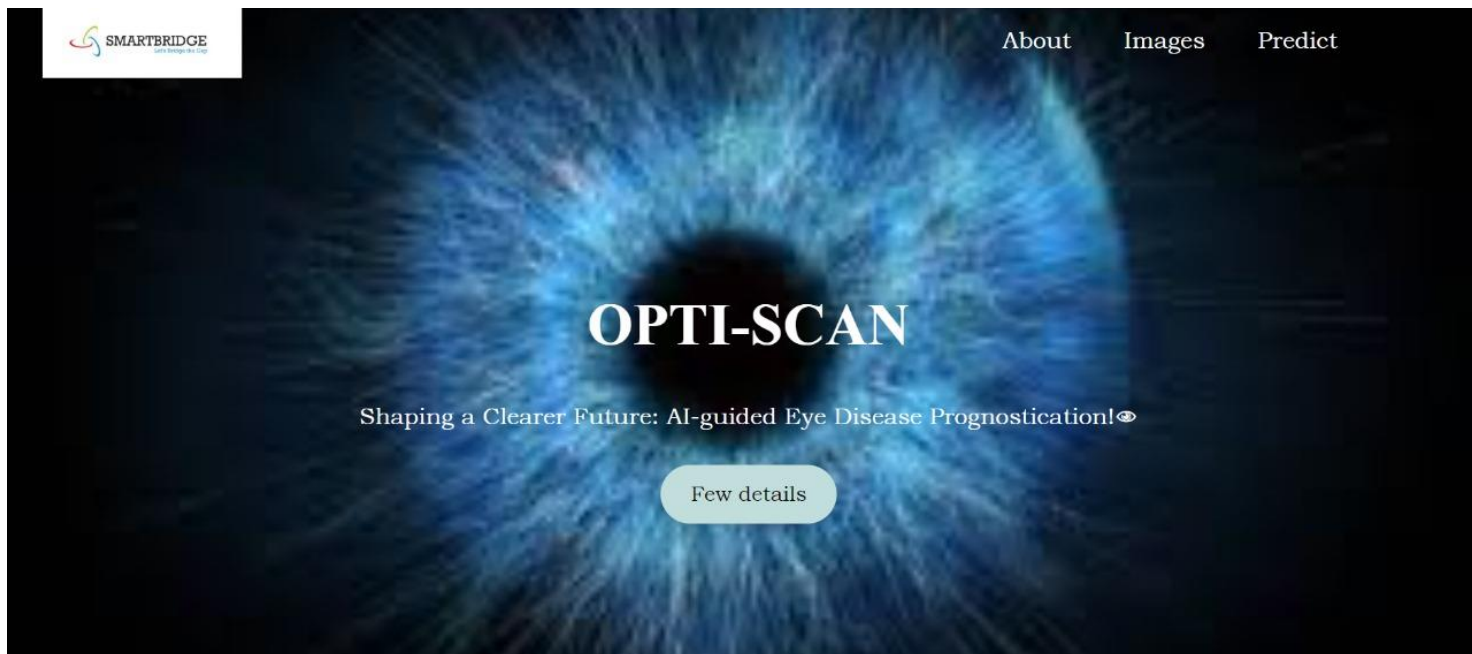
```
if __name__ == "__main__":  
    app.run(debug=False)
```

Activity 3: Run the application

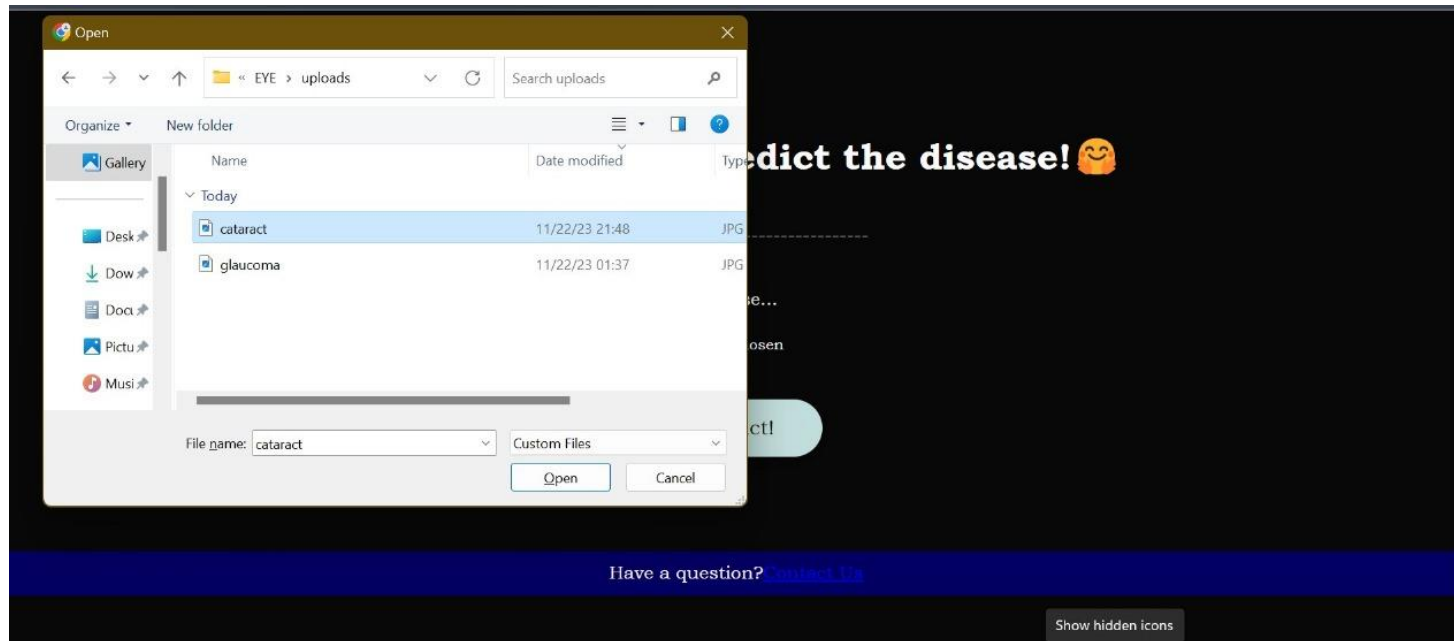
- Open Spyder
- Navigate to the folder where your Python script is.
- Now click on the green play button above.
- Click on the predict button from the top right corner, enter the inputs, click on the Classify button, and see the result/prediction on the web.

```
* Serving Flask app "app" (lazy loading)  
* Environment: production  
  WARNING: This is a development server. Do not use it in a  
production deployment.  
  Use a production WSGI server instead.  
* Debug mode: off  
* Running on http://127.0.0.1:5000/ (Press CTRL+C to quit)
```

The home page looks like this. When you click on the Predict button, you'll be redirected to the predict section



Input 1:

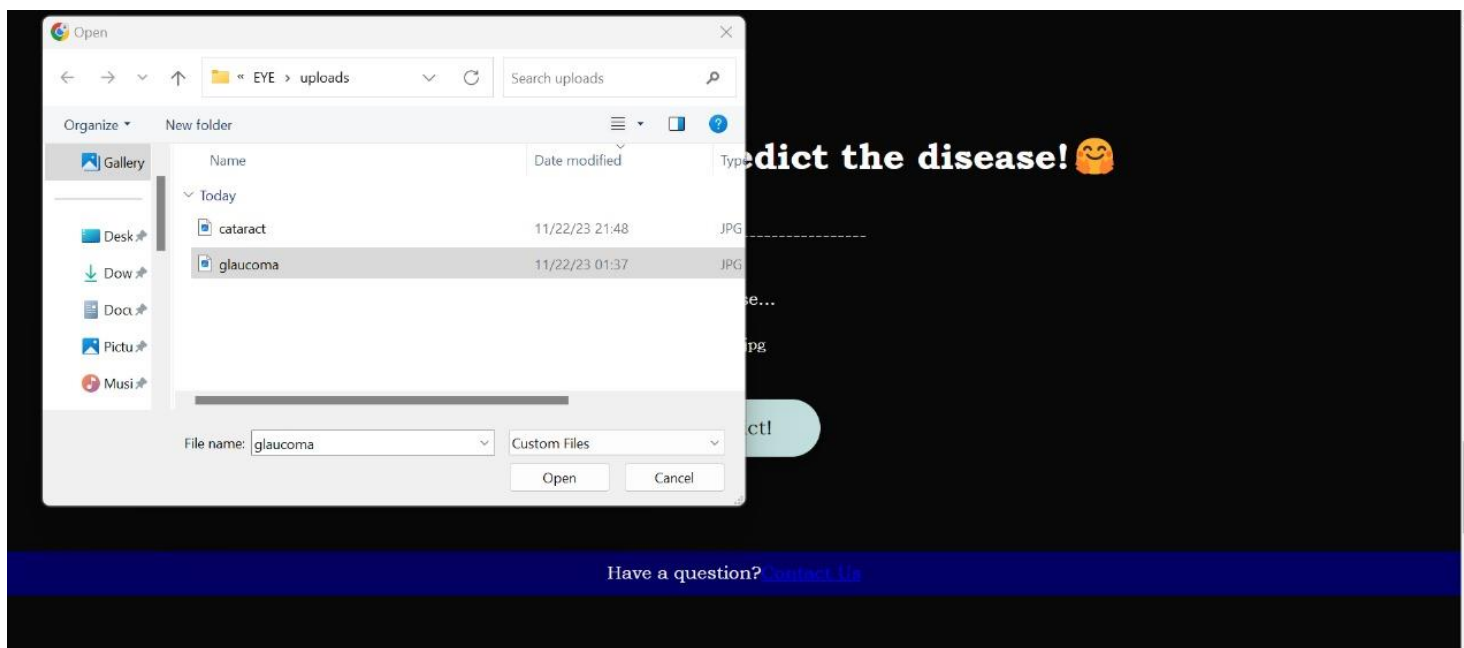


Once you upload the image and click on Predict button, the output will be displayed in the below page

Output 1:



Input 2:



Output2:

