Project Report

Applied Artificial Intelligence

Spring 2024

Department of Software Engineering

Submitted To: Ma'am Shahela Saif

Name	**Roll No**
Hussain Ali Waqar	21I-1123
Sami Irshad	21I-1103
Muhammad Eman	21I-1140

Dated: 5th May 2024

Table Of Contents

Introduction

For this project, we developed an Eye Disease Classification System which can classify eye diseases such as Glaucoma, Diabetic Retinopathy and Cataracts. It can also classify the eyes as being completely Normal.
This classification is done through retinal images.

Project Design

Our project mainly consists of 3 major modules:

1. Frontend
2. Backend
3. Model

The frontend is where the users would be able to interact with the model and use the model to classify retinal images. The users would be able to feed the model a retinal image on which the model would make some prediction.

The backend is where the images uploaded by the users would be stored and sent to the machine learning model.

Finally, the core of the entire project, the model. The machine learning model is responsible for the classification of retinal images and is the most important module of this project.

The Dataset

Before diving into the machine learning model, we had to organize our dataset. The dataset we used consists of different retinal images (a total of 4217 images). The retinal images are organized into separate folders based on their classification such as Normal, Diabetic Retinopathy, Glaucoma and Cataract.

There was no testing, validation or training set. So first we separated the images into training, testing and validation sets. For this purpose, the train_test_split function by scikit-learn library was used.

After running the first cell in the provided “model.ipynb” notebook, our dataset was properly organized into train, test and validation sets.

The train, test and validation folders contain 4 folders each which correspond to the 4 diseases.

Preprocessing the Dataset

Next step was to preprocess the images in the dataset to make them have uniform characteristics.

Standard Image Size

First we added a standard size for all images, i.e 224 x 224. Hence all images that will be fed to the model would have the same standard size. Any size could've been chosen for this purpose and in our case we chose 224 x 224.

Normalization Factor

As images have pixel values ranging from 0-255, we normalized those values to a common range between 0-1 by using a normalization factor. Hence all images now have pixel values in the range 0-1. This further converted our images into a more standardized format which will help train our model in a better way.

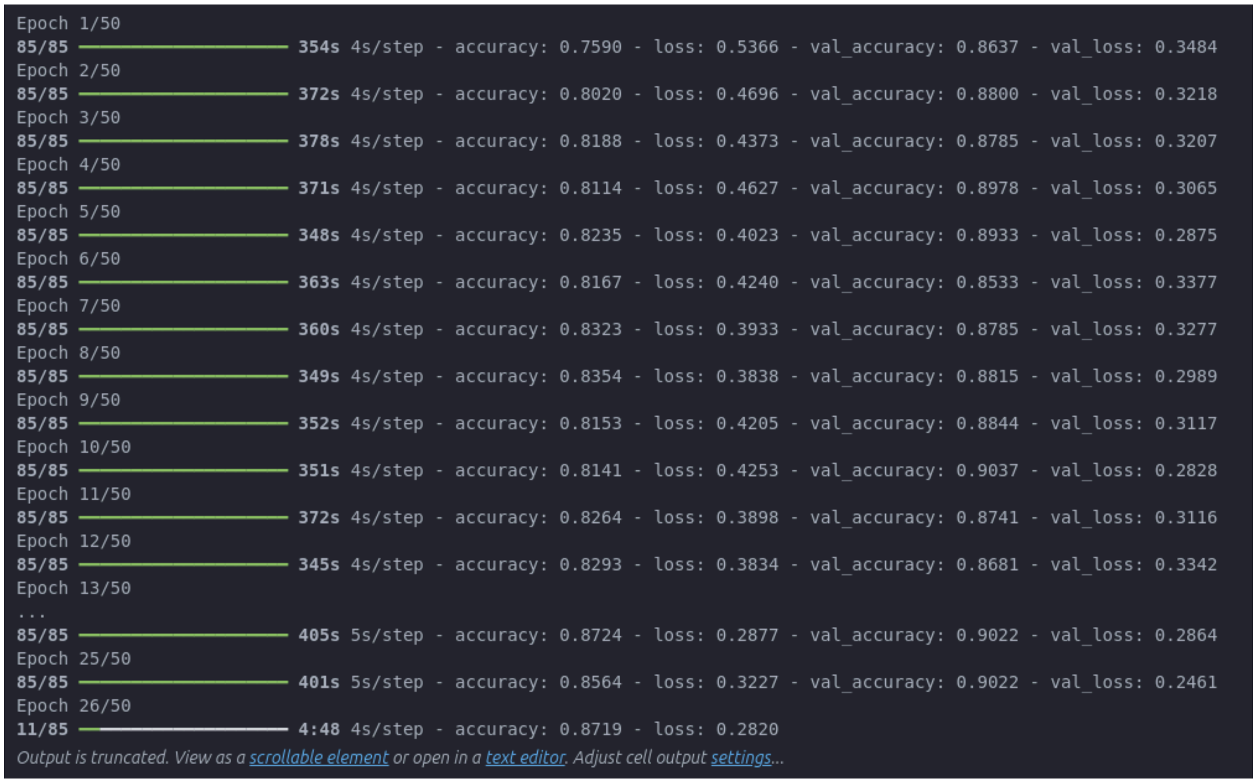
Setting Up the Machine Learning Model:

Next main step is to set up the model before training it. We chose a pre-built model called VGG16. There are other options available as well but we chose VGG16 because it provided us with the best possible accuracy.

For setting up the model, we first froze the layers of the pre-built model so that we don't retrain them in any way. Next we added our own custom layer having 1024 neurons and then another layer called the Dropout layer which is used for preventing overfitting.

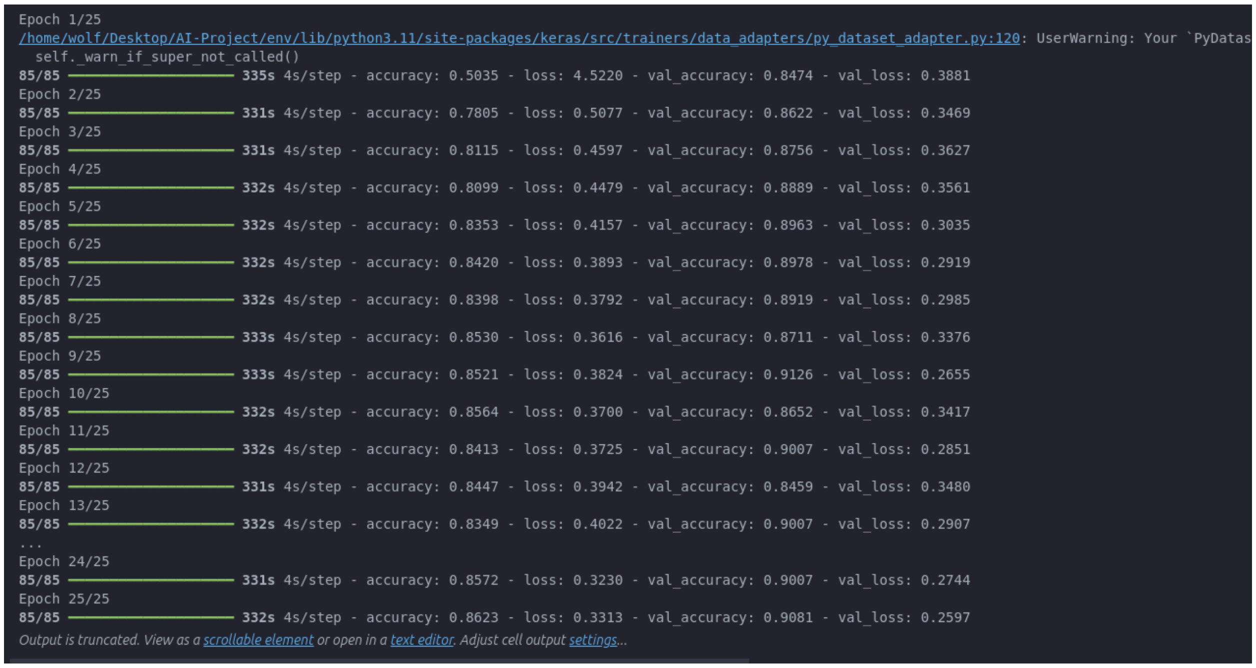
Model Training:

Next, we trained the model. For training we did a number of different experiments. First we trained the VGG16 model on 50 epochs but our computational resources did not allow us to complete the 50 epochs.



```
Epoch 1/50
85/85 ━━━━━━━━━━━━━━━━━━━━ 354s 4s/step - accuracy: 0.7590 - loss: 0.5366 - val_accuracy: 0.8637 - val_loss: 0.3484
Epoch 2/50
85/85 ━━━━━━━━━━━━━━━━━━━━ 372s 4s/step - accuracy: 0.8020 - loss: 0.4696 - val_accuracy: 0.8800 - val_loss: 0.3218
Epoch 3/50
85/85 ━━━━━━━━━━━━━━━━━━━━ 378s 4s/step - accuracy: 0.8188 - loss: 0.4373 - val_accuracy: 0.8785 - val_loss: 0.3207
Epoch 4/50
85/85 ━━━━━━━━━━━━━━━━━━━━ 371s 4s/step - accuracy: 0.8114 - loss: 0.4627 - val_accuracy: 0.8978 - val_loss: 0.3065
Epoch 5/50
85/85 ━━━━━━━━━━━━━━━━━━━━ 348s 4s/step - accuracy: 0.8235 - loss: 0.4023 - val_accuracy: 0.8933 - val_loss: 0.2875
Epoch 6/50
85/85 ━━━━━━━━━━━━━━━━━━━━ 363s 4s/step - accuracy: 0.8167 - loss: 0.4240 - val_accuracy: 0.8533 - val_loss: 0.3377
Epoch 7/50
85/85 ━━━━━━━━━━━━━━━━━━━━ 360s 4s/step - accuracy: 0.8323 - loss: 0.3933 - val_accuracy: 0.8785 - val_loss: 0.3277
Epoch 8/50
85/85 ━━━━━━━━━━━━━━━━━━━━ 349s 4s/step - accuracy: 0.8354 - loss: 0.3838 - val_accuracy: 0.8815 - val_loss: 0.2989
Epoch 9/50
85/85 ━━━━━━━━━━━━━━━━━━━━ 352s 4s/step - accuracy: 0.8153 - loss: 0.4205 - val_accuracy: 0.8844 - val_loss: 0.3117
Epoch 10/50
85/85 ━━━━━━━━━━━━━━━━━━━━ 351s 4s/step - accuracy: 0.8141 - loss: 0.4253 - val_accuracy: 0.9037 - val_loss: 0.2828
Epoch 11/50
85/85 ━━━━━━━━━━━━━━━━━━━━ 372s 4s/step - accuracy: 0.8264 - loss: 0.3898 - val_accuracy: 0.8741 - val_loss: 0.3116
Epoch 12/50
85/85 ━━━━━━━━━━━━━━━━━━━━ 345s 4s/step - accuracy: 0.8293 - loss: 0.3834 - val_accuracy: 0.8681 - val_loss: 0.3342
Epoch 13/50
...
85/85 ━━━━━━━━━━━━━━━━━━━━ 405s 5s/step - accuracy: 0.8724 - loss: 0.2877 - val_accuracy: 0.9022 - val_loss: 0.2864
Epoch 25/50
85/85 ━━━━━━━━━━━━━━━━━━━━ 401s 5s/step - accuracy: 0.8564 - loss: 0.3227 - val_accuracy: 0.9022 - val_loss: 0.2461
Epoch 26/50
11/85 ━━━━━━━━━━━━━━━━━━━━ 4:48 4s/step - accuracy: 0.8719 - loss: 0.2820
Output is truncated. View as a scrollable element or open in a text editor. Adjust cell output settings...
```

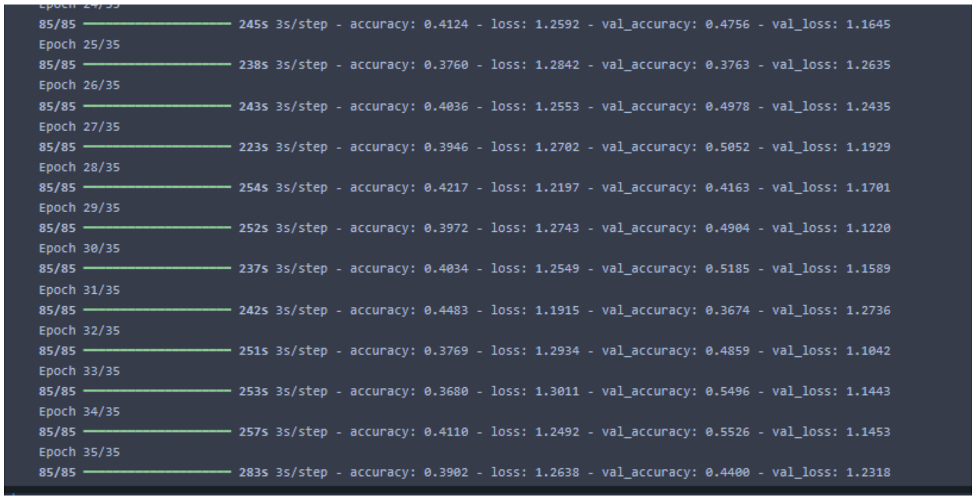
Next, we trained the model again, but this time on 25 epochs and got the following results:



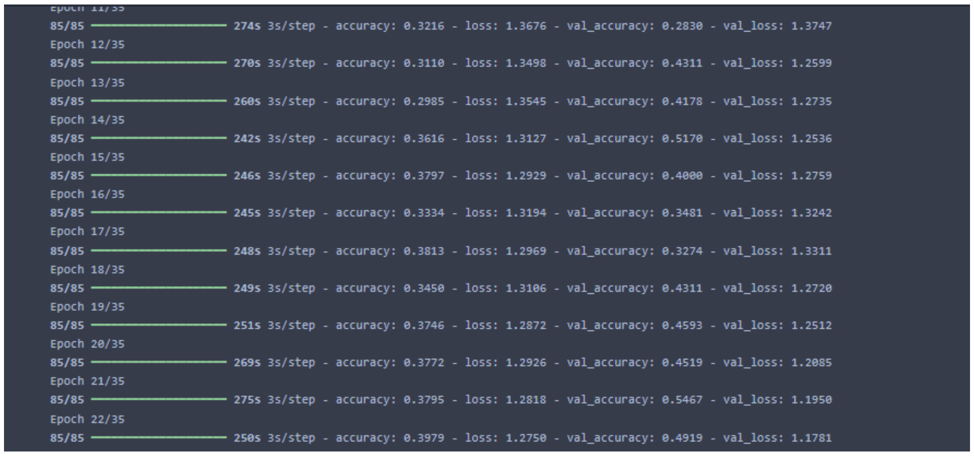
```
Epoch 1/25
/home/wolf/Desktop/AI-Project/env/lib/python3.11/site-packages/keras/src/trainers/data_adapters/py_dataset_adapter.py:120: UserWarning: Your `PyDatas
  self._warn_if_super_not_called()
85/85 ━━━━━━━━━━━━━━━━━━━━ 335s 4s/step - accuracy: 0.5035 - loss: 4.5220 - val_accuracy: 0.8474 - val_loss: 0.3881
Epoch 2/25
85/85 ━━━━━━━━━━━━━━━━━━━━ 331s 4s/step - accuracy: 0.7805 - loss: 0.5077 - val_accuracy: 0.8622 - val_loss: 0.3469
Epoch 3/25
85/85 ━━━━━━━━━━━━━━━━━━━━ 331s 4s/step - accuracy: 0.8115 - loss: 0.4597 - val_accuracy: 0.8756 - val_loss: 0.3627
Epoch 4/25
85/85 ━━━━━━━━━━━━━━━━━━━━ 332s 4s/step - accuracy: 0.8099 - loss: 0.4479 - val_accuracy: 0.8889 - val_loss: 0.3561
Epoch 5/25
85/85 ━━━━━━━━━━━━━━━━━━━━ 332s 4s/step - accuracy: 0.8353 - loss: 0.4157 - val_accuracy: 0.8963 - val_loss: 0.3035
Epoch 6/25
85/85 ━━━━━━━━━━━━━━━━━━━━ 332s 4s/step - accuracy: 0.8420 - loss: 0.3893 - val_accuracy: 0.8978 - val_loss: 0.2919
Epoch 7/25
85/85 ━━━━━━━━━━━━━━━━━━━━ 332s 4s/step - accuracy: 0.8398 - loss: 0.3792 - val_accuracy: 0.8919 - val_loss: 0.2985
Epoch 8/25
85/85 ━━━━━━━━━━━━━━━━━━━━ 333s 4s/step - accuracy: 0.8530 - loss: 0.3616 - val_accuracy: 0.8711 - val_loss: 0.3376
Epoch 9/25
85/85 ━━━━━━━━━━━━━━━━━━━━ 333s 4s/step - accuracy: 0.8521 - loss: 0.3824 - val_accuracy: 0.9126 - val_loss: 0.2655
Epoch 10/25
85/85 ━━━━━━━━━━━━━━━━━━━━ 332s 4s/step - accuracy: 0.8564 - loss: 0.3700 - val_accuracy: 0.8652 - val_loss: 0.3417
Epoch 11/25
85/85 ━━━━━━━━━━━━━━━━━━━━ 332s 4s/step - accuracy: 0.8413 - loss: 0.3725 - val_accuracy: 0.9007 - val_loss: 0.2851
Epoch 12/25
85/85 ━━━━━━━━━━━━━━━━━━━━ 331s 4s/step - accuracy: 0.8447 - loss: 0.3942 - val_accuracy: 0.8459 - val_loss: 0.3480
Epoch 13/25
85/85 ━━━━━━━━━━━━━━━━━━━━ 332s 4s/step - accuracy: 0.8349 - loss: 0.4022 - val_accuracy: 0.9007 - val_loss: 0.2907
...
Epoch 24/25
85/85 ━━━━━━━━━━━━━━━━━━━━ 331s 4s/step - accuracy: 0.8572 - loss: 0.3230 - val_accuracy: 0.9007 - val_loss: 0.2744
Epoch 25/25
85/85 ━━━━━━━━━━━━━━━━━━━━ 332s 4s/step - accuracy: 0.8623 - loss: 0.3313 - val_accuracy: 0.9081 - val_loss: 0.2597
Output is truncated. View as a scrollable element or open in a text editor. Adjust cell output settings...
```

We got 90% accuracy which is satisfactory.

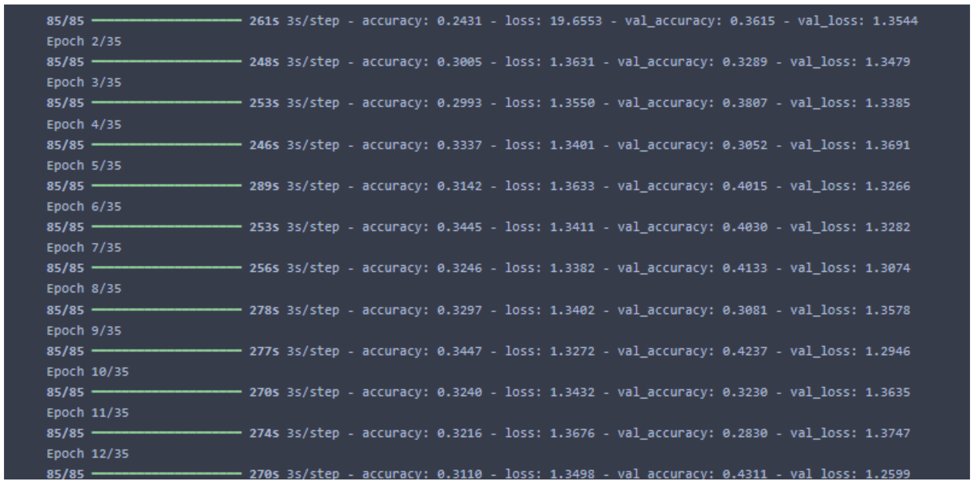
We also tried out another popular model called ResNet50. We got the following results with it:



```
85/85 ━━━━━━━━━━━━━━━━━━━━ 245s 3s/step - accuracy: 0.4124 - loss: 1.2592 - val_accuracy: 0.4756 - val_loss: 1.1645
Epoch 25/35
85/85 ━━━━━━━━━━━━━━━━━━━━ 238s 3s/step - accuracy: 0.3760 - loss: 1.2842 - val_accuracy: 0.3763 - val_loss: 1.2635
Epoch 26/35
85/85 ━━━━━━━━━━━━━━━━━━━━ 243s 3s/step - accuracy: 0.4036 - loss: 1.2553 - val_accuracy: 0.4978 - val_loss: 1.2435
Epoch 27/35
85/85 ━━━━━━━━━━━━━━━━━━━━ 223s 3s/step - accuracy: 0.3946 - loss: 1.2702 - val_accuracy: 0.5052 - val_loss: 1.1929
Epoch 28/35
85/85 ━━━━━━━━━━━━━━━━━━━━ 254s 3s/step - accuracy: 0.4217 - loss: 1.2197 - val_accuracy: 0.4163 - val_loss: 1.1701
Epoch 29/35
85/85 ━━━━━━━━━━━━━━━━━━━━ 252s 3s/step - accuracy: 0.3972 - loss: 1.2743 - val_accuracy: 0.4904 - val_loss: 1.1220
Epoch 30/35
85/85 ━━━━━━━━━━━━━━━━━━━━ 237s 3s/step - accuracy: 0.4034 - loss: 1.2549 - val_accuracy: 0.5185 - val_loss: 1.1589
Epoch 31/35
85/85 ━━━━━━━━━━━━━━━━━━━━ 242s 3s/step - accuracy: 0.4483 - loss: 1.1915 - val_accuracy: 0.3674 - val_loss: 1.2736
Epoch 32/35
85/85 ━━━━━━━━━━━━━━━━━━━━ 251s 3s/step - accuracy: 0.3769 - loss: 1.2934 - val_accuracy: 0.4859 - val_loss: 1.1042
Epoch 33/35
85/85 ━━━━━━━━━━━━━━━━━━━━ 253s 3s/step - accuracy: 0.3680 - loss: 1.3011 - val_accuracy: 0.5496 - val_loss: 1.1443
Epoch 34/35
85/85 ━━━━━━━━━━━━━━━━━━━━ 257s 3s/step - accuracy: 0.4110 - loss: 1.2492 - val_accuracy: 0.5526 - val_loss: 1.1453
Epoch 35/35
85/85 ━━━━━━━━━━━━━━━━━━━━ 283s 3s/step - accuracy: 0.3902 - loss: 1.2638 - val_accuracy: 0.4400 - val_loss: 1.2318
```



```
85/85 ━━━━━━━━━━━━━━━━━━━━ 274s 3s/step - accuracy: 0.3216 - loss: 1.3676 - val_accuracy: 0.2830 - val_loss: 1.3747
Epoch 12/35
85/85 ━━━━━━━━━━━━━━━━━━━━ 270s 3s/step - accuracy: 0.3110 - loss: 1.3498 - val_accuracy: 0.4311 - val_loss: 1.2599
Epoch 13/35
85/85 ━━━━━━━━━━━━━━━━━━━━ 260s 3s/step - accuracy: 0.2985 - loss: 1.3545 - val_accuracy: 0.4178 - val_loss: 1.2735
Epoch 14/35
85/85 ━━━━━━━━━━━━━━━━━━━━ 242s 3s/step - accuracy: 0.3616 - loss: 1.3127 - val_accuracy: 0.5170 - val_loss: 1.2536
Epoch 15/35
85/85 ━━━━━━━━━━━━━━━━━━━━ 246s 3s/step - accuracy: 0.3797 - loss: 1.2929 - val_accuracy: 0.4000 - val_loss: 1.2759
Epoch 16/35
85/85 ━━━━━━━━━━━━━━━━━━━━ 245s 3s/step - accuracy: 0.3334 - loss: 1.3194 - val_accuracy: 0.3481 - val_loss: 1.3242
Epoch 17/35
85/85 ━━━━━━━━━━━━━━━━━━━━ 248s 3s/step - accuracy: 0.3813 - loss: 1.2969 - val_accuracy: 0.3274 - val_loss: 1.3311
Epoch 18/35
85/85 ━━━━━━━━━━━━━━━━━━━━ 249s 3s/step - accuracy: 0.3450 - loss: 1.3106 - val_accuracy: 0.4311 - val_loss: 1.2720
Epoch 19/35
85/85 ━━━━━━━━━━━━━━━━━━━━ 251s 3s/step - accuracy: 0.3746 - loss: 1.2872 - val_accuracy: 0.4593 - val_loss: 1.2512
Epoch 20/35
85/85 ━━━━━━━━━━━━━━━━━━━━ 269s 3s/step - accuracy: 0.3772 - loss: 1.2926 - val_accuracy: 0.4519 - val_loss: 1.2085
Epoch 21/35
85/85 ━━━━━━━━━━━━━━━━━━━━ 275s 3s/step - accuracy: 0.3795 - loss: 1.2818 - val_accuracy: 0.5467 - val_loss: 1.1950
Epoch 22/35
85/85 ━━━━━━━━━━━━━━━━━━━━ 250s 3s/step - accuracy: 0.3979 - loss: 1.2750 - val_accuracy: 0.4919 - val_loss: 1.1781
```

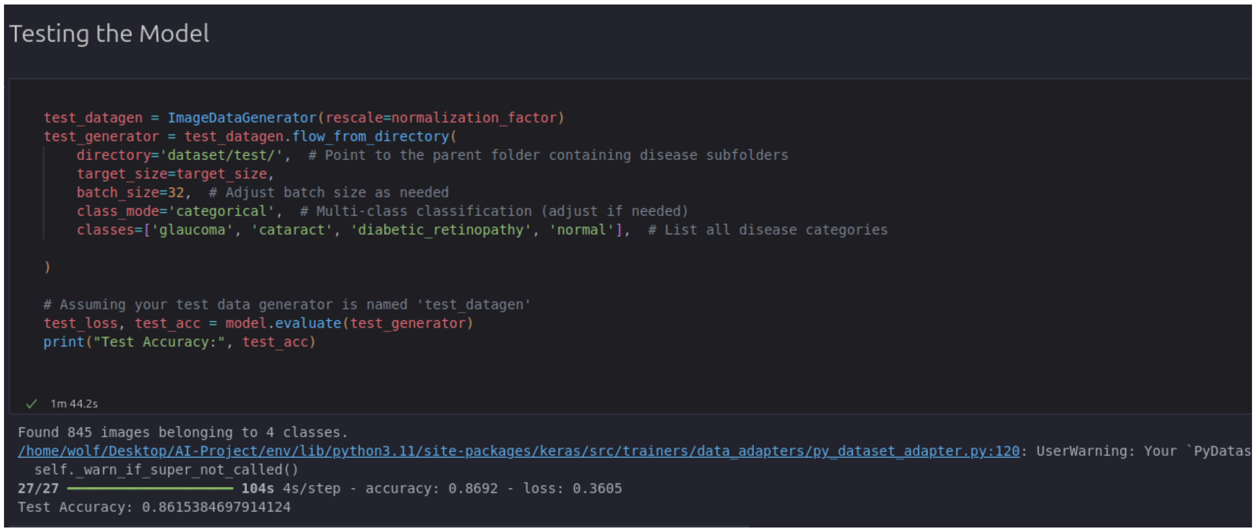


```
85/85 ━━━━━━━━━━━━━━━━━━━━ 261s 3s/step - accuracy: 0.2431 - loss: 19.6553 - val_accuracy: 0.3615 - val_loss: 1.3544
Epoch 2/35
85/85 ━━━━━━━━━━━━━━━━━━━━ 248s 3s/step - accuracy: 0.3005 - loss: 1.3631 - val_accuracy: 0.3289 - val_loss: 1.3479
Epoch 3/35
85/85 ━━━━━━━━━━━━━━━━━━━━ 253s 3s/step - accuracy: 0.2993 - loss: 1.3550 - val_accuracy: 0.3807 - val_loss: 1.3385
Epoch 4/35
85/85 ━━━━━━━━━━━━━━━━━━━━ 246s 3s/step - accuracy: 0.3337 - loss: 1.3401 - val_accuracy: 0.3052 - val_loss: 1.3691
Epoch 5/35
85/85 ━━━━━━━━━━━━━━━━━━━━ 289s 3s/step - accuracy: 0.3142 - loss: 1.3633 - val_accuracy: 0.4015 - val_loss: 1.3266
Epoch 6/35
85/85 ━━━━━━━━━━━━━━━━━━━━ 253s 3s/step - accuracy: 0.3445 - loss: 1.3411 - val_accuracy: 0.4030 - val_loss: 1.3282
Epoch 7/35
85/85 ━━━━━━━━━━━━━━━━━━━━ 256s 3s/step - accuracy: 0.3246 - loss: 1.3382 - val_accuracy: 0.4133 - val_loss: 1.3074
Epoch 8/35
85/85 ━━━━━━━━━━━━━━━━━━━━ 278s 3s/step - accuracy: 0.3297 - loss: 1.3402 - val_accuracy: 0.3081 - val_loss: 1.3578
Epoch 9/35
85/85 ━━━━━━━━━━━━━━━━━━━━ 277s 3s/step - accuracy: 0.3447 - loss: 1.3272 - val_accuracy: 0.4237 - val_loss: 1.2946
Epoch 10/35
85/85 ━━━━━━━━━━━━━━━━━━━━ 270s 3s/step - accuracy: 0.3240 - loss: 1.3432 - val_accuracy: 0.3230 - val_loss: 1.3635
Epoch 11/35
85/85 ━━━━━━━━━━━━━━━━━━━━ 274s 3s/step - accuracy: 0.3216 - loss: 1.3676 - val_accuracy: 0.2830 - val_loss: 1.3747
Epoch 12/35
85/85 ━━━━━━━━━━━━━━━━━━━━ 270s 3s/step - accuracy: 0.3110 - loss: 1.3498 - val_accuracy: 0.4311 - val_loss: 1.2599
```

We got a 39% accuracy even after running for 35 epochs. As it is a large scale model, it requires a lot of time for training and our computational resources didn't allow us for such powerful model training. Hence we settled for VGG16.

Testing The Model:

We tested the model after finalizing it and got the following results:



Testing the Model

```python
test_datagen = ImageDataGenerator(rescale=normalization_factor)
test_generator = test_datagen.flow_from_directory(
    directory='dataset/test/',  # Point to the parent folder containing disease subfolders
    target_size=target_size,
    batch_size=32,  # Adjust batch size as needed
    class_mode='categorical',  # Multi-class classification (adjust if needed)
    classes=['glaucoma', 'cataract', 'diabetic_retinopathy', 'normal'],  # List all disease categories

)

# Assuming your test data generator is named 'test_datagen'
test_loss, test_acc = model.evaluate(test_generator)
print("Test Accuracy:", test_acc)
```

✓ 1m 44.2s

```
Found 845 images belonging to 4 classes.
/home/wolf/Desktop/AI-Project/env/lib/python3.11/site-packages/keras/src/trainers/data_adapters/py_dataset_adapter.py:120: UserWarning: Your `PyDatas
  self._warn_if_super_not_called()
27/27 ━━━━━━━━━━━━━━━━━━━━ 104s 4s/step - accuracy: 0.8692 - loss: 0.3605
Test Accuracy: 0.8615384697914124
```

An 86% accuracy on testing is satisfactory.

Experimentation Results at a Glance:

The following table shows the results obtained by experimenting with the training of models:

Model Name	**Epochs**	**Training Accuracy**	**Validation Accuracy**	**Testing Accuracy**
VGG16	25	86%	90%	86%
ResNet50	35	39%	44%	-

Technologies Used:

The following technologies, libraries have been used for the backend and model training:

- Flask
- NumPy
- Tensorflow

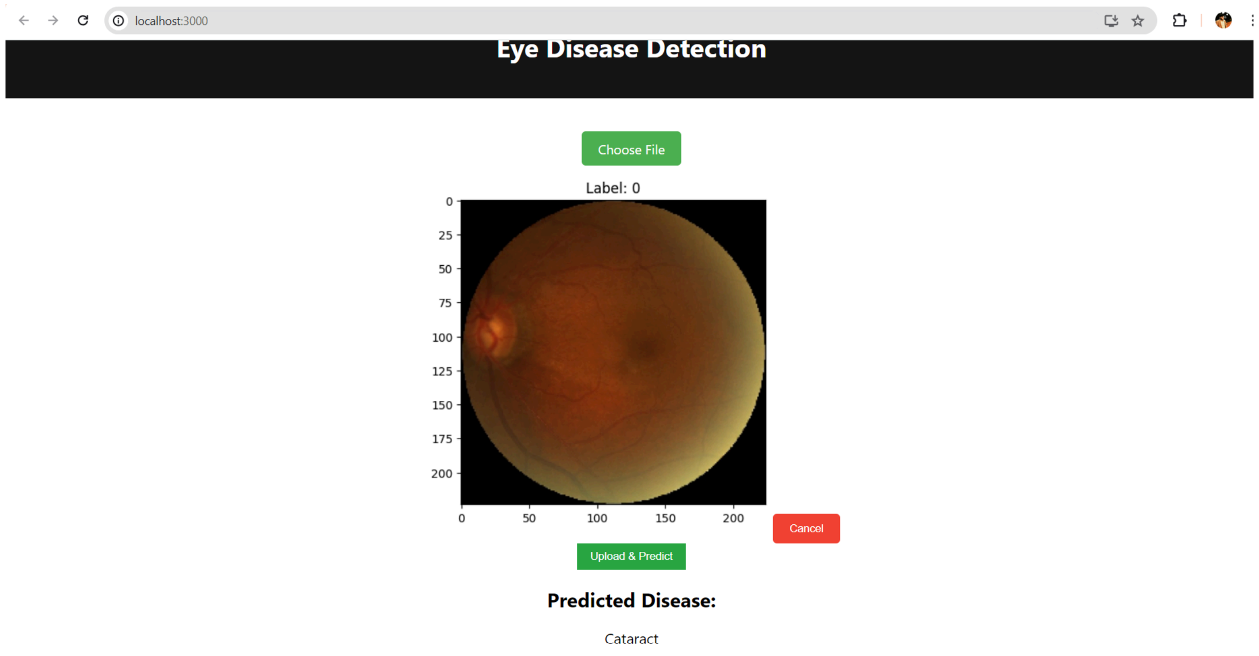
And the following technologies have been used for the frontend:

- ReactJS

Frontend

1. App Component (`App.js`):

- This is the main component of the React application.
- It contains state variables to manage the selected image, predicted disease, error messages, and the name of the disease.
- The **`handleImageChange`** function is called when a user selects an image file. It updates the `selectedImage` state with the chosen file.
- The **`handleCancel`** function clears the selected image and resets other states when the user clicks the cancel button.
- The **`handleUpload`** function is called when the user clicks the "Upload & Predict" button. It sends a POST request to the backend with the selected image using the Fetch API.
- Upon receiving a response from the backend, if the response is successful (`response.ok`), it updates the `predictedDisease` state with the predicted class index received from the backend and calls the `getDiseaseName` function to convert the index into the corresponding disease name, which is then set in the `diseaseName` state.
- The UI displays the selected image, the predicted disease (if available), and any error messages.



Predicted Disease:

Cataract

Backend (Flask):

1. Flask App (`app.py`):
- This is the main backend Flask application.
- It defines an endpoint `/classify` to receive POST requests containing images to be classified.
- When an image is uploaded, it preprocesses the image, makes predictions using the pre-trained model, and returns the predicted class index to the frontend.
- The predicted class index is sent as the response to the frontend.

Integration:

1. Communication between Frontend and Backend:
- The frontend and backend communicate over HTTP.
- When the user uploads an image and clicks "Upload & Predict," the frontend sends a POST request to the backend's `/classify` endpoint with the selected image.
- The backend receives the image, processes it, makes predictions, and sends the predicted class index back to the frontend.
- The frontend receives the predicted class index, converts it into a disease name, and displays it to the user.

2. Cross-Origin Resource Sharing (CORS):
- To allow communication between the frontend and backend running on different origins (localhost:3000 and localhost:5000), CORS headers are set in the Flask application to permit requests from the frontend origin.
- This prevents the CORS-related error that occurs when attempting to make requests between different origins.

3. Preprocessing and Prediction:
- The uploaded image is preprocessed to match the input requirements of the pre-trained model.
- The preprocessed image is passed through the model, which makes predictions regarding the presence of eye diseases.
- The predicted class index is returned to the frontend for display.

4. Displaying Results:
- The frontend displays the selected image and the predicted disease name (if available) to the user.
- If there is an error during image processing or prediction, an error message is displayed instead.

In summary, the frontend allows users to upload images, while the backend processes these images, makes predictions, and sends the results back to the frontend for display. This interaction enables users to detect eye diseases using the application.