DL in Applied Mathematics

Lecture 11: Computer Vision: Image Classification with TensorFlow and Keras

Marat Nurtas
PhD in Mathematical and Computer Modeling
Department of Mathematical and Computer Modeling
International Information Technology University, Almaty, Kazakhstan

Mastering Image Classification with TensorFlow and Keras

A Comprehensive Guide to CNNs and Machine Learning in Python

If you’re curious about how machine learning algorithms can classify images with precision and accuracy, you’ve come to the right place.

In this *lecture*, we’ll delve deep into the world of image classification, exploring the fundamentals of Convolutional Neural Networks (CNNs).

Image classification is a fascinating field that lies at the intersection of computer vision and machine learning.

It enables machines to recognize and categorize objects, patterns, and features within images, opening doors to a wide range of applications, from medical diagnostics to autonomous vehicles.

In this lecture , we’ll explore the intricate workings of CNNs and their role in identifying and classifying images.

We’ll be using TensorFlow and Keras that are the most popular frameworks for building deep learning models, to develop robust and efficient image classification systems.

We’ll cover essential topics such as data augmentation, transfer learning, and model evaluation.

Dataset

We will be using eye disease dataset for this tutorial. To obtain the eye disease classification dataset.

You can access it from Kaggle, a popular platform for data science competitions, datasets, and machine learning resources.

Import Libraries

```
# import system libs
import os
import time
# import data handling tools
import cv2
import numpy as np
import pandas as pd
from PIL import Image
import matplotlib.pyplot as plt
from sklearn.model_selection import train_test_split
from sklearn.metrics import confusion_matrix, classification_report, f1_score
```

```python
# import Deep learning Libraries
import tensorflow as tf
from tensorflow import keras
from tensorflow.keras.models import Model
from tensorflow.keras.metrics import categorical_crossentropy
from tensorflow.keras.preprocessing.image import
ImageDataGenerator
from tensorflow.keras.layers import Flatten, Dense, Activation,
GlobalAveragePooling2D

# Ignore Warnings
import warnings
warnings.filterwarnings("ignore")
```

Class for Loading and Splitting Datasets

```
class EyeDiseaseDataset:
def __init__(self, dataDir):
self.data_dir = dataDir

def dataPaths(self):
filepaths = []
labels = []
folds = os.listdir(self.data_dir)
for fold in folds:
foldPath = os.path.join(self.data_dir, fold)
filelist = os.listdir(foldPath)
for file in filelist:
fpath = os.path.join(foldPath, file)
filepaths.append(fpath)
labels.append(fold)
return filepaths, labels
```

```
def dataFrame(self, files, labels):
Fseries = pd.Series(files, name='filepaths')
Lseries = pd.Series(labels, name='labels')
return pd.concat([Fseries, Lseries], axis=1)

def split_(self):
files, labels = self.dataPaths()
df = self.dataFrame(files, labels)
strat = df['labels']
trainData, dummyData = train_test_split(df, train_size=0.8, shuffle=True,
random_state=42, stratify=strat)
strat = dummyData['labels']
validData, testData = train_test_split(dummyData, train_size=0.5,
shuffle=True, random_state=42, stratify=strat)
return trainData, validData, testData
```

This Python code defines a class named EyeDiseaseDataset that helps us to manage and preprocess dataset of eye disease images.

- __init__(self, dataDir): The constructor initializes the object with the directory path dataDir, which is the root directory containing the dataset.
- dataPaths(self): This method traverses the directory structure of the dataset and collects file paths and corresponding labels. It iterates through each fold (subdirectory) within the dataset directory, collects file paths, and assigns labels based on the fold names.

- dataFrame(self, files, labels): Constructs a Pandas DataFrame from the collected file paths and labels, creating two Series: one for file paths and another for labels.
- split_(self): This method splits the dataset into three subsets: training, validation, and testing sets. It uses the train_test_split function from Scikit-Learn to split the data while ensuring stratified sampling based on the labels.
- It first splits the data into 80% training and 20% dummy data.
- Then, it splits the dummy data into 50% validation and 50% testing sets.

The method returns three Data Frames: trainData, validData, and testData, each containing file paths and corresponding labels for their respective subsets.

```
dataDir='/content/dataset/’
```

```
dataSplit = EyeDiseaseDataset(dataDir)
train_data, valid_data, test_data = dataSplit.split_()
```

- dataDir='/content/dataset/': This is a directory path where the dataset is located. It's specified as '/content/dataset/', indicating that the dataset is stored in the 'dataset' folder.
- dataSplit = EyeDiseaseDataset(dataDir): Here, we initialize an instance of the EyeDiseaseDataset class, passing the dataset directory path as an argument.
- train_data, valid_data, test_data = dataSplit.split_(): Then we split the dataset into three subsets: training, validation, and testing data. The split_()

Function for Data Augmentation

```
def augment_data( train_df, valid_df, test_df, batch_size=32):
img_size = (256,256)
channels = 3
color = 'rgb'

train_datagen = tf.keras.preprocessing.image.ImageDataGenerator(
rotation_range=30,
horizontal_flip=True,
vertical_flip=True,
brightness_range=[0.5, 1.5])
valid_test_datagen = tf.keras.preprocessing.image.ImageDataGenerator()
```

```
train_generator = train_datagen.flow_from_dataframe(
train_df,
x_col='filepaths',
y_col='labels',
target_size=img_size,
color_mode=color,
batch_size=batch_size,
shuffle=True,
class_mode='categorical’
)
print("Shape of augmented training images:",
train_generator.image_shape)
```

```
valid_generator = valid_test_datagen.flow_from_dataframe(
valid_df,
x_col='filepaths',
y_col='labels',
target_size=img_size,
color_mode=color,
batch_size=batch_size,
shuffle=True,
class_mode='categorical'
)

print("Shape of validation images:", valid_generator.image_shape)
```

```python
test_generator = valid_test_datagen.flow_from_dataframe(
test_df,
x_col='filepaths',
y_col='labels',
target_size=img_size,
color_mode=color,
batch_size=batch_size,
shuffle=False,
class_mode='categorical'
)

print("Shape of test images:", test_generator.image_shape)

return train_generator, valid_generator, test_generator
```

The function augment_data takes three DataFrames (train_df, valid_df, and test_df) containing file paths and labels of images.

It generates augmented image data using TensorFlow's ImageDataGenerator and returns corresponding data generators for training, validation, and testing.

Parameters:

- train_df: DataFrame containing file paths and labels for training images.
- valid_df: DataFrame containing file paths and labels for validation images.
- test_df: DataFrame containing file paths and labels for testing images.
- batch_size: Batch size for training, validation, and testing data generators (default is 32).

Image Augmentation:

- rotation_range: Range for random rotations applied to images (here, 30 degrees).

- horizontal_flip and vertical_flip: Boolean indicating if random horizontal and vertical flips should be applied.

- brightness_range: Range for adjusting brightness of images.

Data Generators:

- Three data generators are created using ImageDataGenerator.flow_from_dataframe the method, one each for training, validation, and testing.
- Each generator is configured with specific parameters:
- target_size: Target size for images after resizing.
- color_mode: Color mode for images ('rgb' for red-green-blue).
- batch_size: Number of images in each batch.
- shuffle: Whether to shuffle the data after each epoch.
- class_mode: Type of label array generated ('categorical' for categorical labels).

Output:

- The function prints the shape of augmented training, validation, and testing images.
- It returns three data generators: train_generator, valid_generator, and test_generator, each containing augmented image data for their respective sets.

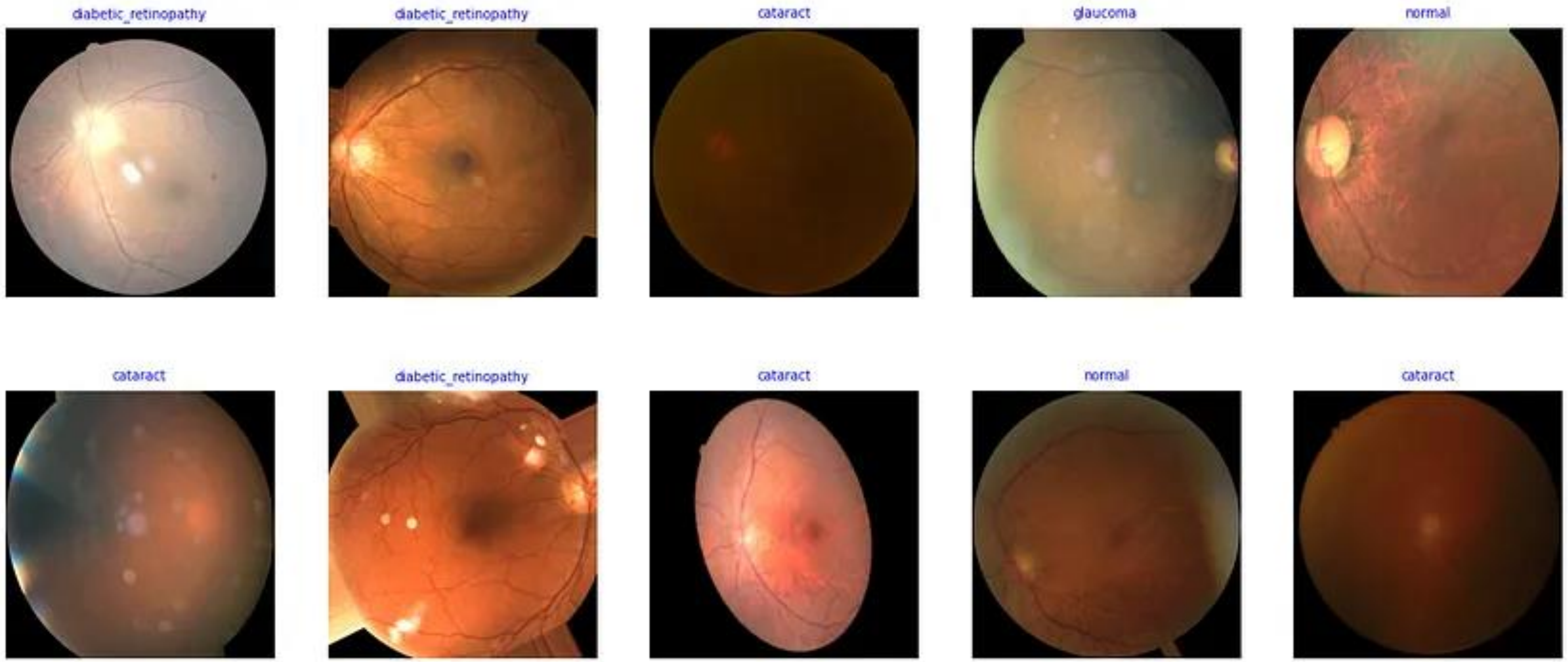
```
train_augmented, valid_augmented, test_augmented = 
augment_data(train_data, valid_data, test_data)
```

Display Augmented images

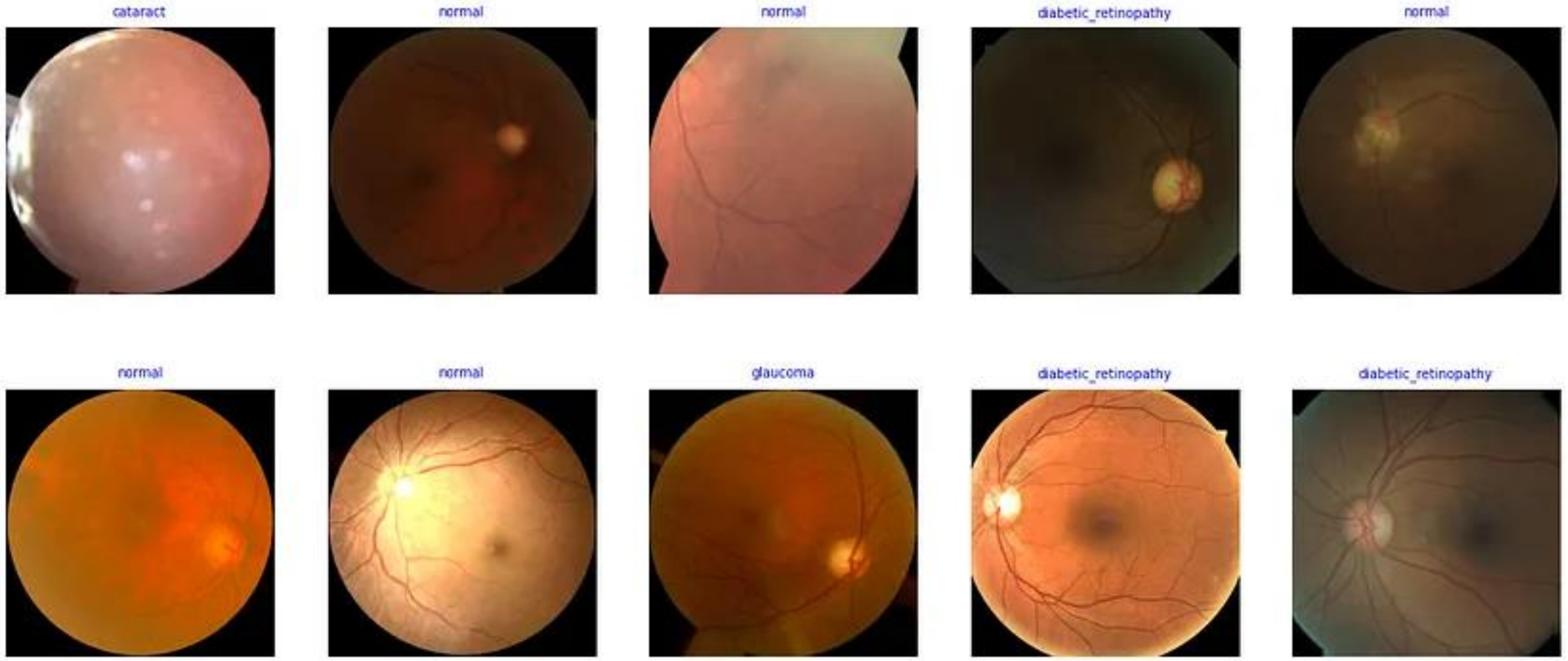
```
def show_images(gen):

g_dict = gen.class_indices
 # defines dictionary {'class': index}
classes = list(g_dict.keys())
# defines list of dictionary's kays (classes), classes names : string
images, labels = next(gen)
# get a batch size samples from the generator
```

```
 length = len(labels)
   sample = min(length, 20)
   plt.figure(figsize= (15, 17))
   for i in range(sample):
       plt.subplot(5, 5, i + 1)
       image = images[i] / 255
       plt.imshow(image)
       index = np.argmax(labels[i])
       class_name = classes[index]
       plt.title(class_name, color= 'blue', fontsize= 7 )
       plt.axis('off')
   plt.show()
show_images(train_augmented)
```

Augmented Images



Augmented Images



Download and compile the model

```
from tensorflow.keras.applications import EfficientNetB3

classes = len(list(train_augmented.class_indices.keys()))

base_model = EfficientNetB3(weights='imagenet', include_top=False,
input_shape=(256, 256, 3))

for layer in base_model.layers:
layer.trainable = False
x = base_model.output
x = GlobalAveragePooling2D()(x)
x = Dense(512, activation='relu')(x)
```

```
predictions = Dense(classes, activation='softmax')(x)

model = Model(inputs=base_model.input, outputs=predictions)

optimizer = tf.keras.optimizers.Adam(learning_rate=0.001)

model.compile(optimizer=optimizer, loss='categorical_crossentropy',
metrics=['accuracy'])
```

Here we define a convolutional neural network (CNN) model using transfer learning with EfficientNetB3 architecture pre-trained on ImageNet. The model is then compiled for training.

Classes Calculation:

The number of classes is determined by extracting the keys of the class indices from the train_augmented data generator. The number of classes corresponds to the number of unique labels in the training set.

Base Model Definition:

- The EfficientNetB3 architecture pre-trained on ImageNet is instantiated using EfficientNetB3(weights='imagenet', include_top=False, input_shape=(512, 512, 3)).

- include_top=False excludes the fully connected layers at the top of the network, allowing for custom classification layers to be added.
- input_shape=(512, 512, 3) specifies the input shape of the images.

Freezing Base Model Layers:

- The layers of the pre-trained base model are set to non-trainable using a loop over `base_model.layers`.
- This ensures that weights in the pre-trained layers are not updated during training.

Model Customization:

- The output of the base model is passed through a global average pooling layer to reduce spatial dimensions.
- A fully connected layer with 512 units and ReLU activation is added for feature extraction.
- The final layer is a Dense layer with softmax activation, producing class probabilities for the classification task.

Model Compilation:

- The model is compiled using the Adam optimizer with a learning rate of 0.001.
- Categorical cross-entropy is chosen as the loss function, since it’s a multi-class classification problem.
- Accuracy is used as the evaluation metric during training.

Fit the model

```
history = model.fit(
train_augmented,
epochs=10, #you can train the model for more epochs
validation_data=valid_augmented,
)
```

Plot the Accuracy and Loss

```
train_accuracy = history.history['accuracy']
val_accuracy = history.history['val_accuracy']
print("Training Accuracy:", train_accuracy[-1])
print("Validation Accuracy:", val_accuracy[-1])
plt.plot(history.history['loss'], label='Training Loss')
plt.plot(history.history['val_loss'], label='Validation Loss')
plt.xlabel('Epoch')
plt.ylabel('Loss')
plt.legend()
plt.show()
plt.plot(history.history['accuracy'], label='Training Accuracy')
plt.plot(history.history['val_accuracy'], label='Validation Accuracy')
plt.xlabel('Epoch')
plt.ylabel('Accuracy')
plt.legend()
plt.show()
```

Training and Validation Accuracy:

The model achieved a final training accuracy of train_accuracy[-1] and a validation accuracy of val_accuracy[-1] .

These metrics provide insights into how well the model learned from the training data and its ability to generalize to unseen validation data.

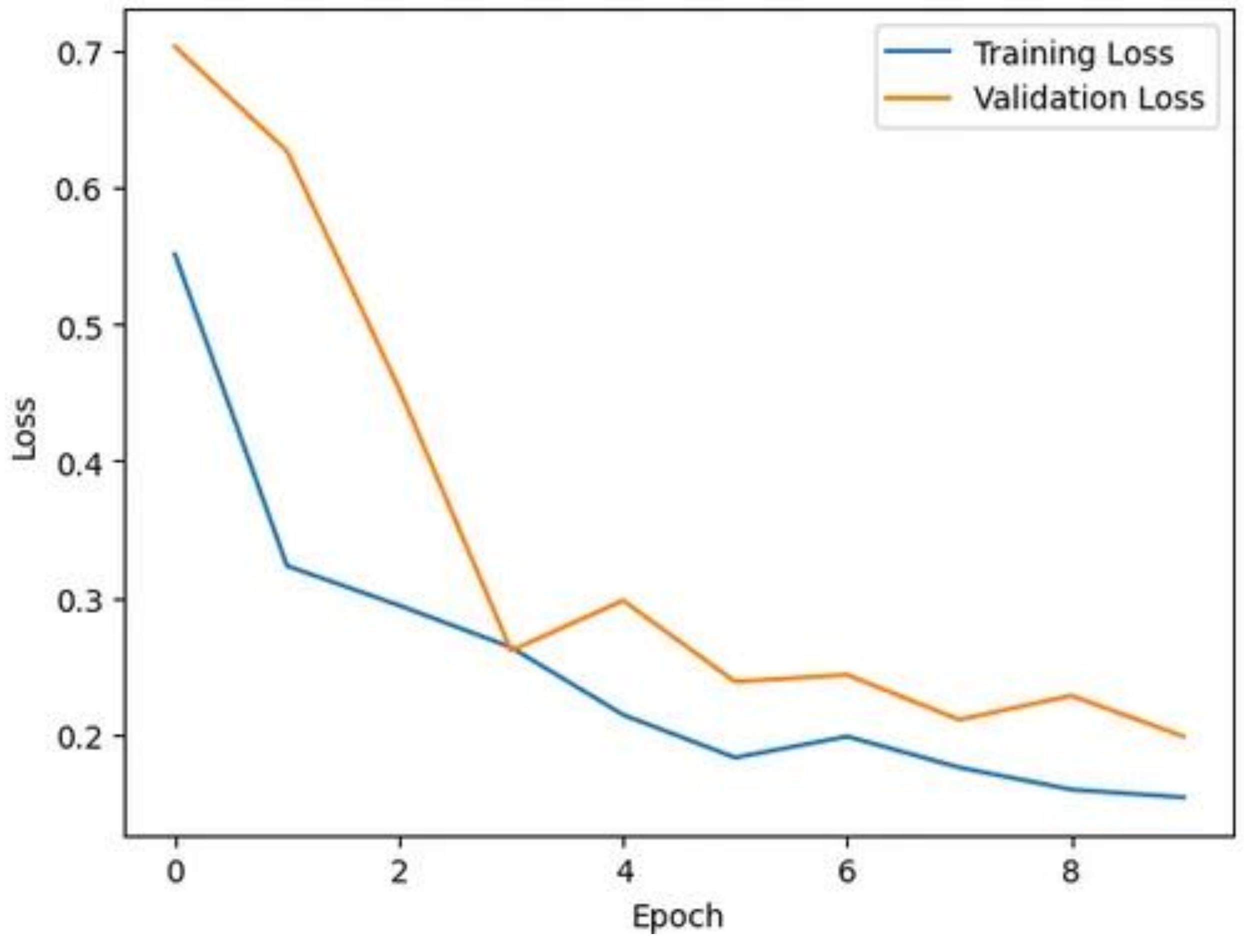
Loss Curve Analysis: By plotting the training and validation loss curves over epochs, we observed the trend of loss reduction during training.

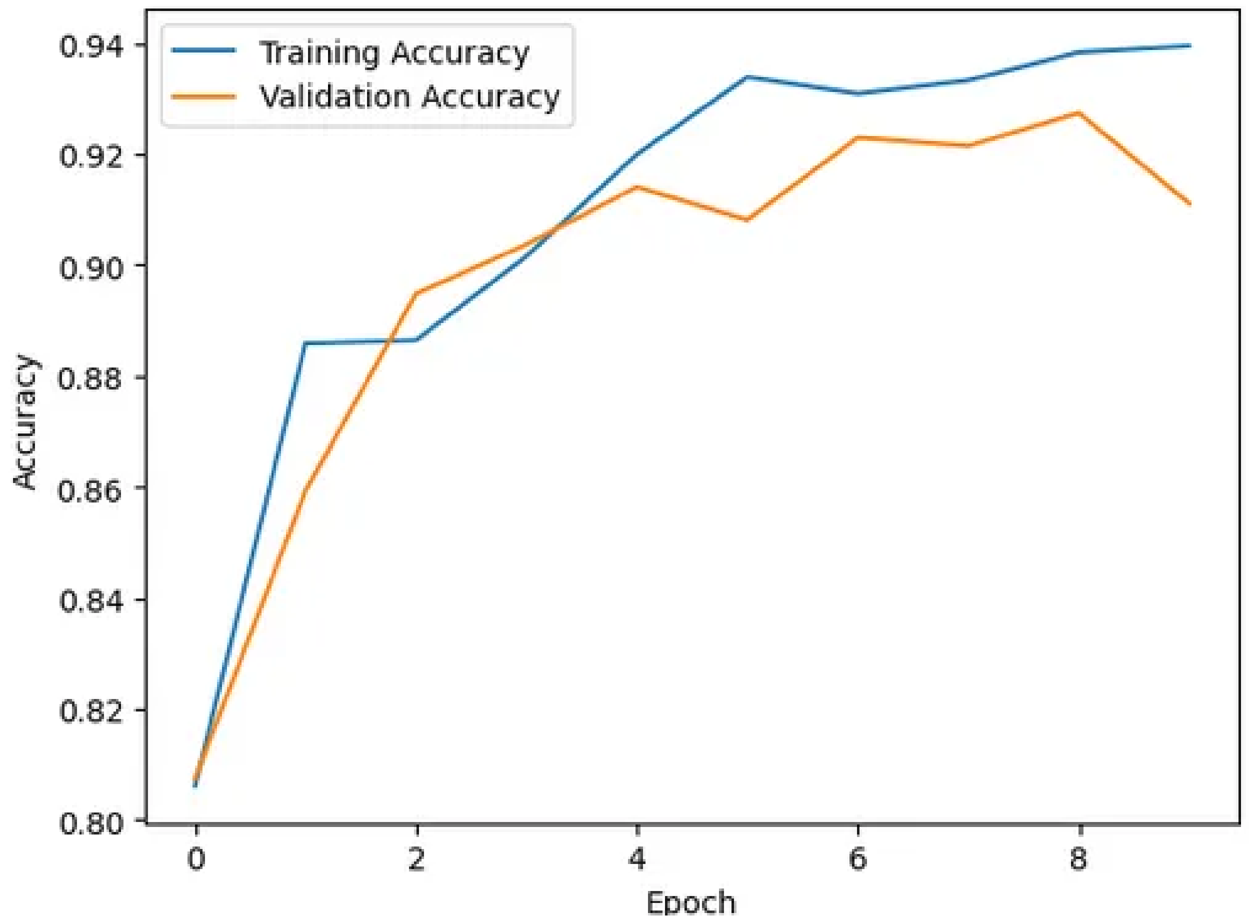
The convergence of training and validation loss indicates the effectiveness of the model’s learning process and its ability to minimize prediction errors.

Accuracy Curve Analysis: The accuracy curves plotted against epochs demonstrated the model’s improvement in correctly classifying images over the training and validation datasets.

A consistent increase in accuracy suggests successful model learning and adaptation to the dataset characteristics.

Plots:


Training Loss
Validation Loss
0.7
0.6
0.5
0.4
0.3
0.2
Loss
0
2
4
6
8
Epoch

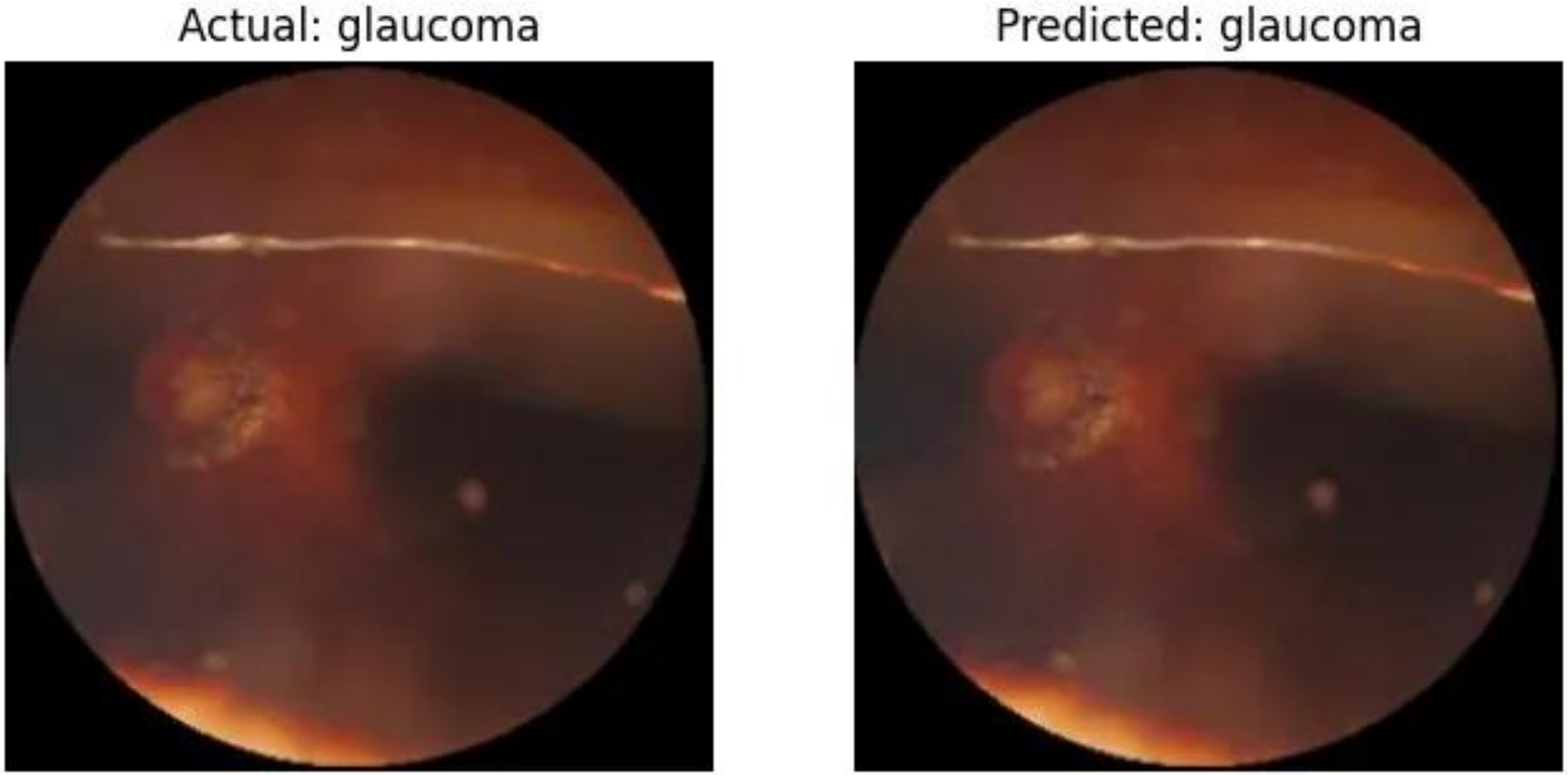

Training Accuracy
Validation Accuracy
0.94
0.92
0.90
0.88
0.86
0.84
0.82
0.80
Accuracy
0
2
4
6
8
Epoch

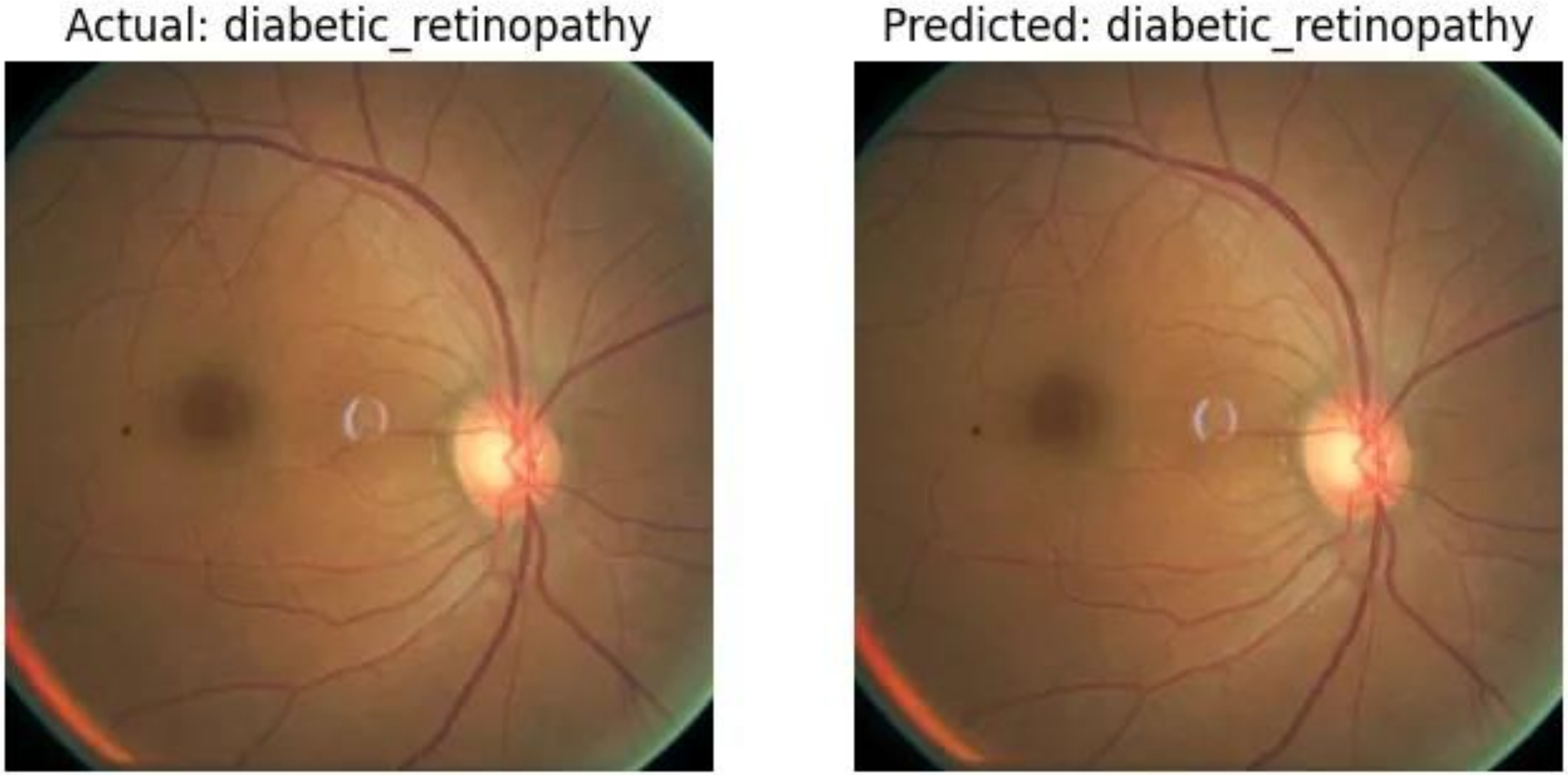
Display the Actual and Predicted images

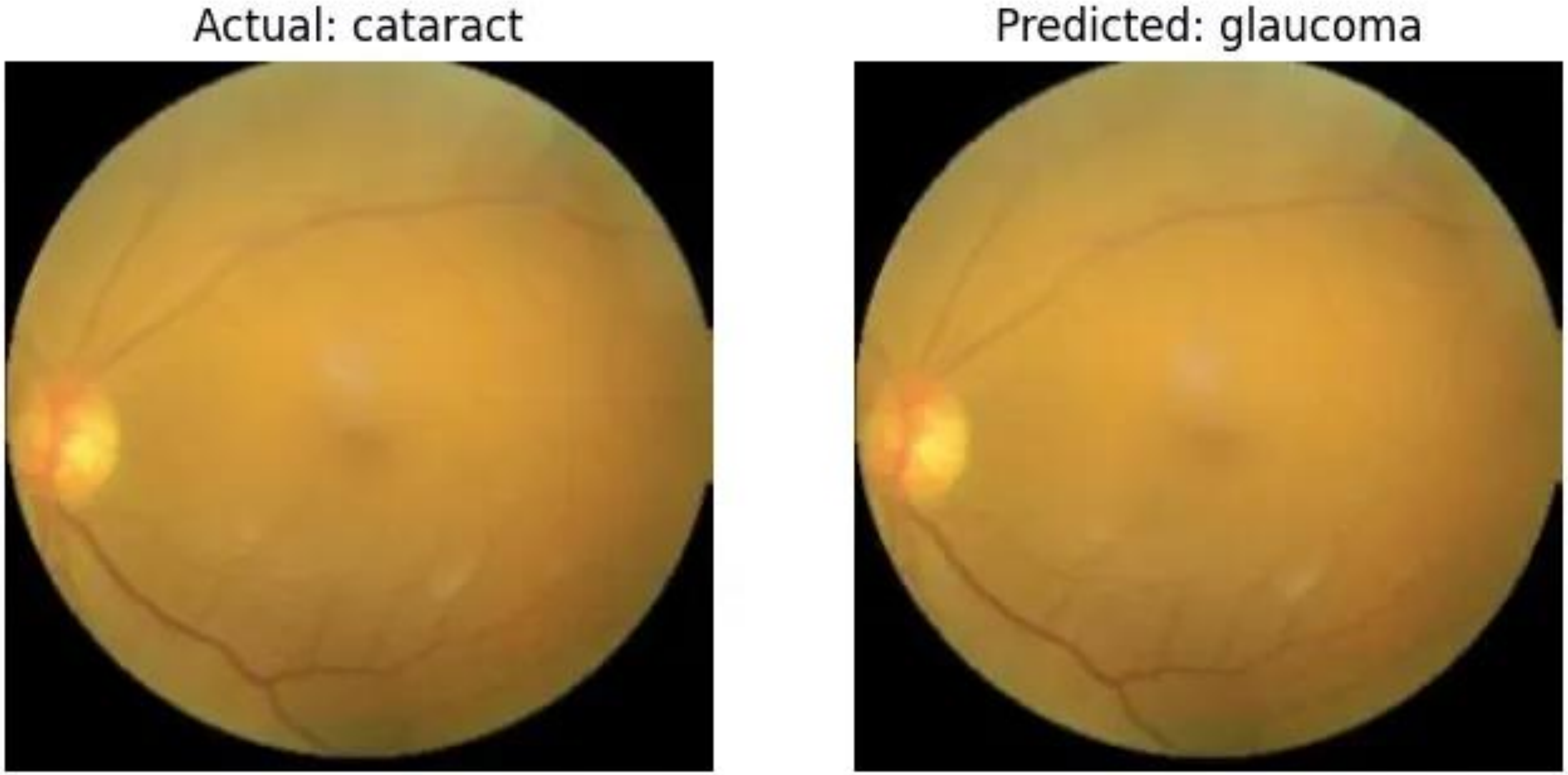
```
def plot_actual_vs_predicted(model, test_data, num_samples=3):
# Get a batch of test data
test_images, test_labels = next(iter(test_data))

predictions = model.predict(test_images)

class_labels = list(train_augmented.class_indices.keys())
```

```
sample_indices = np.random.choice(range(len(test_images)),
num_samples, replace=False)
# Plot the images with actual and predicted labels
for i in sample_indices:
actual_label = class_labels[np.argmax(test_labels[i])]
predicted_label = class_labels[np.argmax(predictions[i])]
plt.figure(figsize=(8, 4))

# Actual Image
plt.subplot(1, 2, 1)
plt.imshow(test_images[i].astype(np.uint8))
plt.title(f'Actual: {actual_label}')
plt.axis('off')
```

```
# Predicted Image
plt.subplot(1, 2, 2)
plt.imshow(test_images[i].astype(np.uint8))
plt.title(f'Predicted: {predicted_label}')
plt.axis('off')
plt.show()

plot_actual_vs_predicted(model, test_augmented)
```


Actual: glaucoma
Predicted: glaucoma


Actual: diabetic_retinopathy
Predicted: diabetic_retinopathy


Actual: cataract
Predicted: glaucoma

And that’s it for our look into eye disease classification models! I hope you’ve found these insights helpful in understanding how machine learning can help with eye conditions.

By learning about how these models work, we’re all one step closer to improving eye care for everyone.

As we move forward, let’s remember that making healthcare better is something we can all contribute to.

By using the tools and ideas we’ve talked about, we can work together to make eye care even better for people everywhere.

Thank you for attention!