"Diabetic Retinopathy Detection Using Artificial Intelligence"

Course Objectives

In this course, we are going to focus on the following learning objectives:

1. Understand the theory and intuition behind Deep Neural Networks, Residual Nets, and Convolutional Neural Networks (CNNs).
2. Apply Python libraries to import, pre-process and visualize images.
3. Perform data augmentation to improve model generalization capability.
4. Build a deep learning model based on Convolutional Neural Network and Residual blocks using Keras with Tensorflow 2.0 as a backend.
5. Compile and fit Deep Learning model to training data.
6. Assess the performance of trained CNN and ensure its generalization using various KPIs such as accuracy, precision and recall.

Project Structure

The hands on project on ***"Diabetic Retinopathy Detection Using Artificial Intelligence"*** is divided into following tasks:

Task #1: Understand the Problem Statement and Business Case

Task #2: Import Libraries and Datasets

Task #3: Perform Data Exploration and Visualization

Task #4: Perform Data Augmentation and Create Data Generator

Task #5: Understand the Theory and Intuition Behind Convolutional Neural Networks

Task #6: Build a ResNet Deep Neural Network Model

Task #7: Compile and Train the Deep Neural Network Model

Task #8: Assess the Performance of the Trained Model

Understand the Problem Statement and Business Case

Understanding the problem and its real-world context is a crucial initial step in any data science or machine learning project.

Problem Statement:

Diabetic retinopathy is a diabetes-related eye disease that can lead to blindness if not detected and treated early. The goal of this project is to develop an artificial intelligence (AI) model that can accurately detect diabetic retinopathy in retinal images. This detection process involves the analysis of retinal images to identify signs of the disease, which can then be used for early diagnosis and timely medical intervention.

Business Case:

The business case for this project is based on addressing the following key aspects:

1. Medical Significance: Diabetic retinopathy is a common complication of diabetes and one of the leading causes of blindness in adults. Early detection and intervention can prevent vision loss. Therefore, developing an AI tool for diabetic retinopathy detection can significantly impact public health.

2. Access to Specialists: Many regions lack access to eye care specialists, leading to delayed diagnoses. AI-based detection can bridge this gap by providing an initial assessment even in underserved areas.

3. Efficiency: Automating the screening process with AI can reduce the time and cost of manual evaluations by healthcare professionals, making healthcare more efficient and cost-effective.

4. Scalability: Once developed, an AI model can be easily scaled and deployed across various healthcare facilities and locations, reaching a broader population.

5. Data Availability: The availability of large datasets of retinal images labeled with diabetic retinopathy severity makes it feasible to develop a machine learning model for this task.

6. Diagnostic Accuracy: AI models have the potential to achieve high diagnostic accuracy, minimizing the risk of both false positives and false negatives in diabetic retinopathy detection.

Project Objectives:

The project's primary objectives are as follows:

1. Data Preparation: Preprocess and prepare the dataset of retinal images, including data augmentation and splitting into training and testing sets.

2. Model Building: Develop a deep learning model, specifically a Residual Neural Network (ResNet), for diabetic retinopathy detection.

3. Training and Validation: Train the model using the prepared dataset while monitoring its performance during training.

4. Evaluation: Evaluate the model's performance on a separate testing dataset using appropriate metrics.

5. Model Deployment: Deploy the model, allowing it to be used for diabetic retinopathy detection in real-world applications.

6. Accessibility and Interpretability: Ensure that the model is accessible to healthcare professionals and that its decisions are interpretable.

By addressing these objectives, the project aims to create a valuable tool for the early detection of diabetic retinopathy, contributing to improved healthcare outcomes for diabetic patients. Additionally, the project emphasizes the importance of transparency, accuracy, and ethical considerations in AI-based medical applications.

Import Libraries/Datasets

1. Import pandas as pd:

- In this line, we import the Pandas library and alias it as 'pd.' Pandas is a widely-used data manipulation library in Python, and we use it to handle and analyze data efficiently.

2. Import numpy as np:

- Here, we import the NumPy library and alias it as 'np.' NumPy is another fundamental library for numerical computing in Python, providing support for arrays and mathematical functions.

3. Import tensorflow as tf:

- TensorFlow is a popular open-source machine learning framework. By importing it as 'tf,' we can use its functions and classes for building and training machine learning models.

4. From tensorflow import keras:

- TensorFlow's 'keras' module is a high-level deep learning API. We import it to access Keras functions for building and training neural networks.

5. Import os:

- The 'os' module provides a way to interact with the operating system. In this context, it will be used for file operations and directory handling.

6. Import matplotlib.pyplot as plt:

- Matplotlib is a data visualization library in Python. We import its 'pyplot' module and alias it as 'plt' to create visualizations and plots.

7. Import PIL:

- PIL stands for Python Imaging Library. It is used to work with images in Python, and it allows us to open, manipulate, and save image files.

8. Import seaborn as sns:

- Seaborn is a data visualization library built on top of Matplotlib. We import it and alias it as 'sns' to create more appealing and informative statistical graphics.

9. From sklearn.model_selection import train_test_split:

- Scikit-learn (sklearn) is a machine learning library for Python. We import the 'train_test_split' function to split our dataset into training and testing sets for model evaluation.

10. From sklearn.utils import shuffle:

- We import the 'shuffle' function from scikit-learn's 'utils' module. It's used to shuffle the dataset, which can be helpful for preventing bias during training.

11. From tensorflow.keras.preprocessing.image import ImageDataGenerator:

- This line imports the 'ImageDataGenerator' class from Keras. It is used for data augmentation, which involves creating variations of the training images to improve model generalization.

12. From tensorflow.keras.applications.resnet50 import ResNet50:

- ResNet50 is a pre-trained deep learning model. We import it from Keras' applications to use it as a base model or feature extractor.

13. From tensorflow.keras.applications.inception_resnet_v2 import InceptionResNetV2:

- InceptionResNetV2 is another pre-trained deep learning model available in Keras. We import it to use as a base model for feature extraction or transfer learning.

*14. From tensorflow.keras.layers import *:*

- In this line, we import all layers from Keras. This allows us to use various neural network layers in building our custom models.

15. From tensorflow.keras.models import Model, load_model:

- We import the 'Model' class and 'load_model' function from Keras. 'Model' is used to create a custom neural network architecture, and 'load_model' is used to load pre-trained models.

16. From tensorflow.keras.initializers import glorot_uniform:

- 'glorot_uniform' is an initialization technique used for neural network weights. We import it to set the initial values of model weights.

17. From tensorflow.keras.utils import plot_model:

- We import the 'plot_model' function, which is used to create visual representations of neural network architectures.

18. From IPython.display import display:

- 'IPython.display' is a library that enables rich media output in Jupyter notebooks. We import the 'display' function for showing images and plots directly in the notebook.

19. From tensorflow.keras import backend as K:

- Keras backend provides an interface to low-level operations of TensorFlow. We import it and alias it as 'K' for specific operations that might need it.

20. From tensorflow.keras.optimizers import SGD:

- We import the SGD (Stochastic Gradient Descent) optimizer from Keras. SGD is a common optimization algorithm used to train deep learning models.

21. From tensorflow.keras.preprocessing.image import ImageDataGenerator:

- This line imports the 'ImageDataGenerator' class once again. Data augmentation is a crucial step in many deep learning projects, so we import it again for convenience.

22. From tensorflow.keras.models import Model, Sequential:

- We import the 'Model' and 'Sequential' classes from Keras. 'Model' is used for creating custom models with multiple inputs or outputs, and 'Sequential' is used for simpler, linear models.

23. From tensorflow.keras.callbacks import ReduceLROnPlateau, EarlyStopping, ModelCheckpoint, LearningRateScheduler:

- Callbacks are functions that can be applied during model training. Here, we import several useful callbacks for model training and optimization.

These imports set up the essential libraries and tools required for this project, which will be used in subsequent tasks.

Inspecting Dataset Directory: Before working with the dataset, we explore the contents of the dataset directory to understand its structure. We use os.listdir to list the subdirectories, which correspond to different classes of retinopathy images. This is an essential step to become familiar with the data's organization.

```
os.listdir("C:\\Users\\user\\Desktop\\train")

# Output: ['Mild', 'Moderate', 'No_DR', 'Proliferate_DR', 'Severe']
```

Accessing Subdirectory: We access one of the subdirectories, in this case, the 'Mild' class, to further understand the structure of the dataset. This step ensures that we can navigate and access the images within the subdirectories correctly.

```
os.listdir(os.path.join("C:\\Users\\user\\Desktop\\train", "C:\\Users\\user\\Desktop\\train\\Mild"))
```

Preparing the Data: To build a deep learning model, we need to prepare the dataset. We create two empty lists, train and label, which will store the image file paths and their corresponding class labels. We iterate through each class directory and extract image paths while keeping track of their labels.

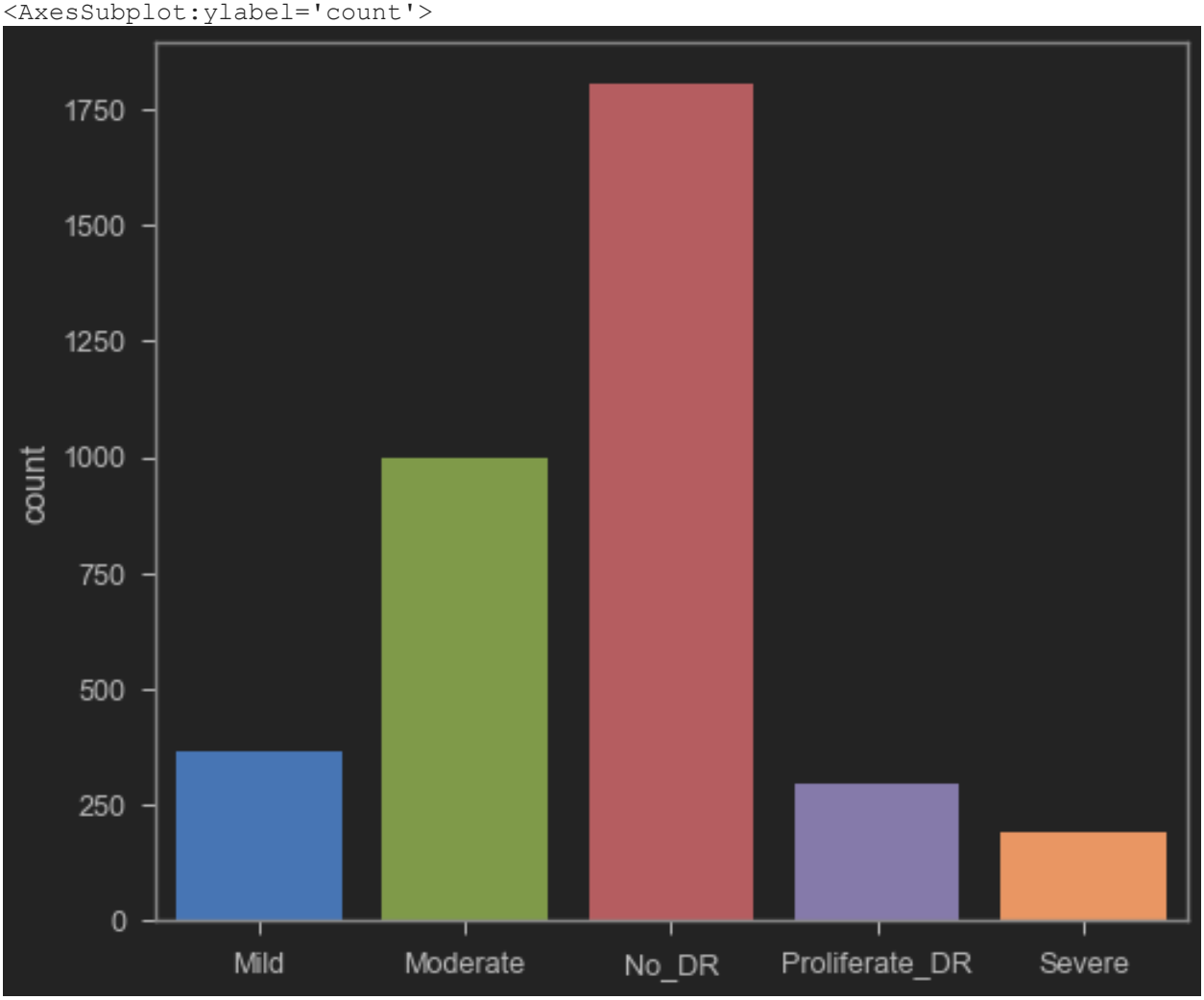
```
train = []

label = []


# os.listdir returns the list of files in the folder, in this case image class names

for i in os.listdir("C:\\Users\\user\\Desktop\\train"):

  train_class = os.listdir(os.path.join("C:\\Users\\user\\Desktop\\train", i))

  for j in train_class:

    img = os.path.join("C:\\Users\\user\\Desktop\\train", i, j)

    train.append(img)

    label.append(i)


print('Number of train images : {} \n'.format(len(train)))


# Output: Number of train images: 3660
```

Data Inspection: Finally, we inspect the contents of the train and label lists to ensure that we have successfully captured the image file paths and their corresponding labels.

Data Visualization: To gain insights into the distribution of the classes, we visualize the number of images in each class using a count plot from the `seaborn` library.

This comprehensive import dataset process provides an overview of how we accessed, organized, and prepared the dataset for the Diabetic Retinopathy Detection project. These steps are crucial for building and training deep learning models on image data.

```
C:\Users\user\anaconda3\lib\site-packages\seaborn\_decorators.py:36:
FutureWarning: Pass the following variable as a keyword arg: x. From version
0.12, the only valid positional argument will be `data`, and passing other
arguments without an explicit keyword will result in an error or
misinterpretation.
  warnings.warn(
<AxesSubplot:ylabel='count'>
```



PERFORM DATA EXPLORATION AND DATA VISUALIZATION

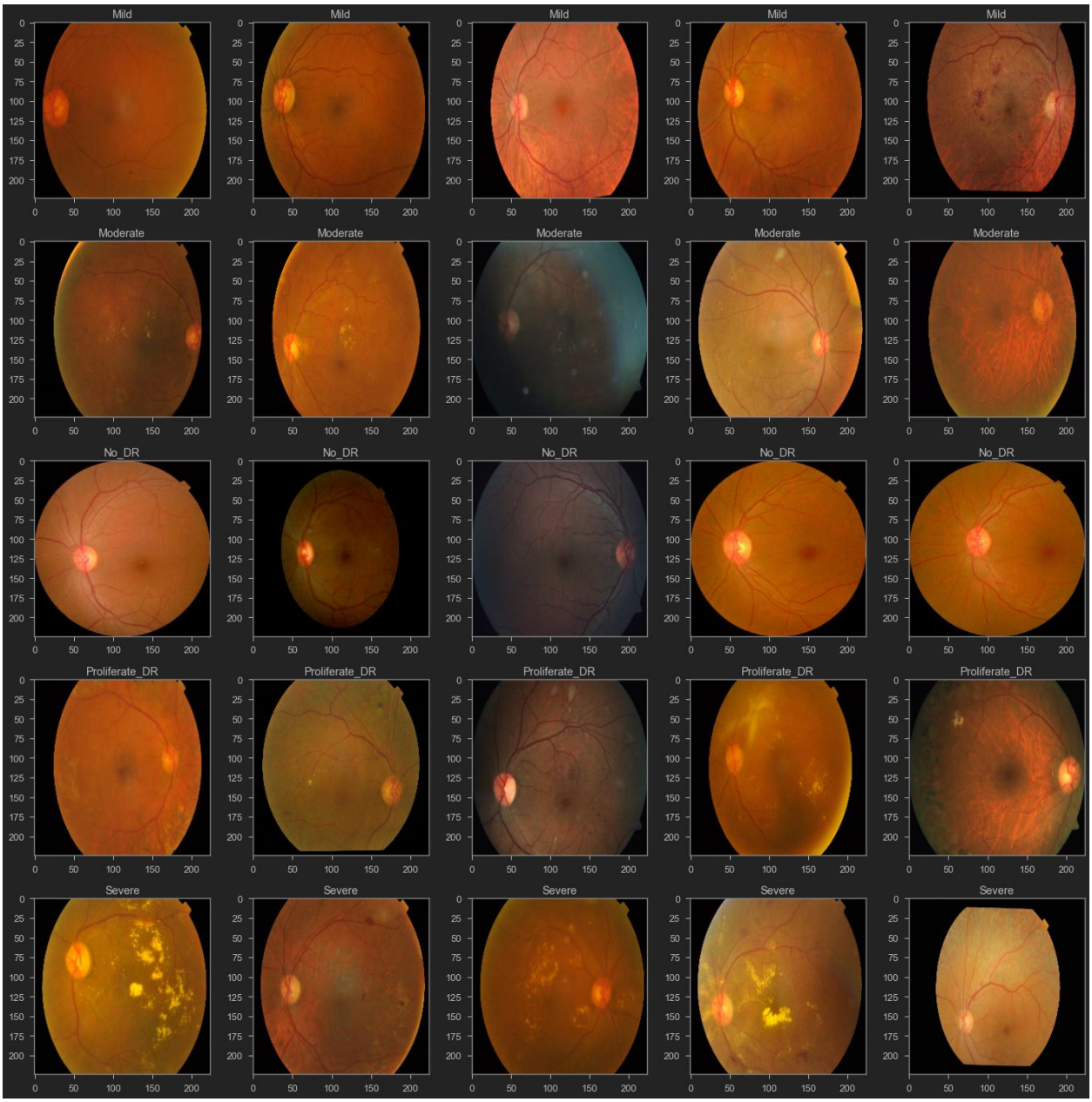
We perform data exploration and visualization to gain a better understanding of the dataset for Diabetic Retinopathy Detection. The primary goals are to visualize images from different classes and assess the dataset's characteristics. Here's how we achieved this:

Visualizing Images from Each Class: To understand what the images look like in each class, we plot five images from each class. The images are displayed in a 5x5 grid, and each class is represented by a row of images.

```
fig, axs = plt.subplots(5, 5, figsize=(20, 20))

count = 0

for i in os.listdir("C:\\Users\\user\\Desktop\\train"):

    # Get the list of images in a given class

    train_class = os.listdir(os.path.join("C:\\Users\\user\\Desktop\\train", i))

    # Plot 5 images per class

    for j in range(5):

        img = os.path.join("C:\\Users\\user\\Desktop\\train", i, train_class[j])

        img = PIL.Image.open(img)

        axs[count][j].title.set_text(i)

        axs[count][j].imshow(img)

    count += 1

fig.tight_layout()
```

In the code above, we loop through the class directories and retrieve image paths to display in the grid. Each row of images represents a different class, allowing us to visually inspect the dataset.

This data exploration and visualization step provides an initial look at the images in each class, helping us understand the dataset's diversity and content. It's an essential part of any computer vision project, as it aids in identifying potential challenges and opportunities for preprocessing and model building.



Checking the Number of Images in Each Class

We analyze the dataset's class distribution by checking the number of images in each class. This step is essential for understanding the dataset's balance and can provide insights into potential class-related challenges.

Here are the details of the analysis:

1. Counting Images in Each Class: We loop through the directories in the training dataset and count the number of images in each class using the `os.listdir` function. For each class, we access its subdirectory and count the files.

```python
No_images_per_class = []
Class_name = []
for i in os.listdir("C:\\Users\\user\\Desktop\\train"):
  train_class = os.listdir(os.path.join("C:\\Users\\user\\Desktop\\train", i))
  No_images_per_class.append(len(train_class))
  Class_name.append(i)
  print('Number of images in {} = {} \n'.format(i, len(train_class)))
```

The results of this analysis are as follows:

- Number of images in Mild = 368

- Number of images in Moderate = 999

- Number of images in No_DR = 1805

- Number of images in Proliferate_DR = 295

- Number of images in Severe = 193

These statistics provide valuable information about the class distribution. In particular, we can see that the "No_DR" class has significantly more images compared to other classes, while "Proliferate_DR" and "Severe" have fewer images. Understanding the distribution of classes can help in making informed decisions during model training and evaluation.

Creating a DataFrame for the Dataset

Here we create a Pandas DataFrame to organize and represent the dataset. The DataFrame includes two main columns: "Image" and "Labels," where each row corresponds to an image sample from the dataset.

Here are the details of this step:

1. Creating the DataFrame: We create the DataFrame using Pandas, and it consists of two columns:

- "Image": This column contains the file path of each image in the dataset, which allows us to access the images.

- "Labels": This column contains the class labels corresponding to each image, indicating the category or class to which the image belongs.

2. Example Entries: We display a few example entries from the DataFrame to showcase its structure. Each row contains an image's file path and its associated class label. This organization makes it easier to manage and manipulate the dataset for further analysis and model training.

```python
retina_df = pd.DataFrame({'Image': train, 'Labels': label})
retina_df
```

The resulting DataFrame consists of 3660 rows, each representing an image sample. This structured format simplifies data exploration, visualization, and further data preprocessing steps as part of the project.

Exploring Class Distribution

To understand how many images belong to each class and visualize this distribution.

Here are the details of this step:

1. Counting Images per Class: To determine the number of images in each class, we iterate through the classes (e.g., 'Mild,' 'Moderate,' 'No_DR,' 'Proliferate_DR,' 'Severe') and count the number of images within each class. The results are stored in two lists:

- `No_images_per_class`: This list contains the count of images for each class.

- `Class_name`: This list holds the names of the classes.

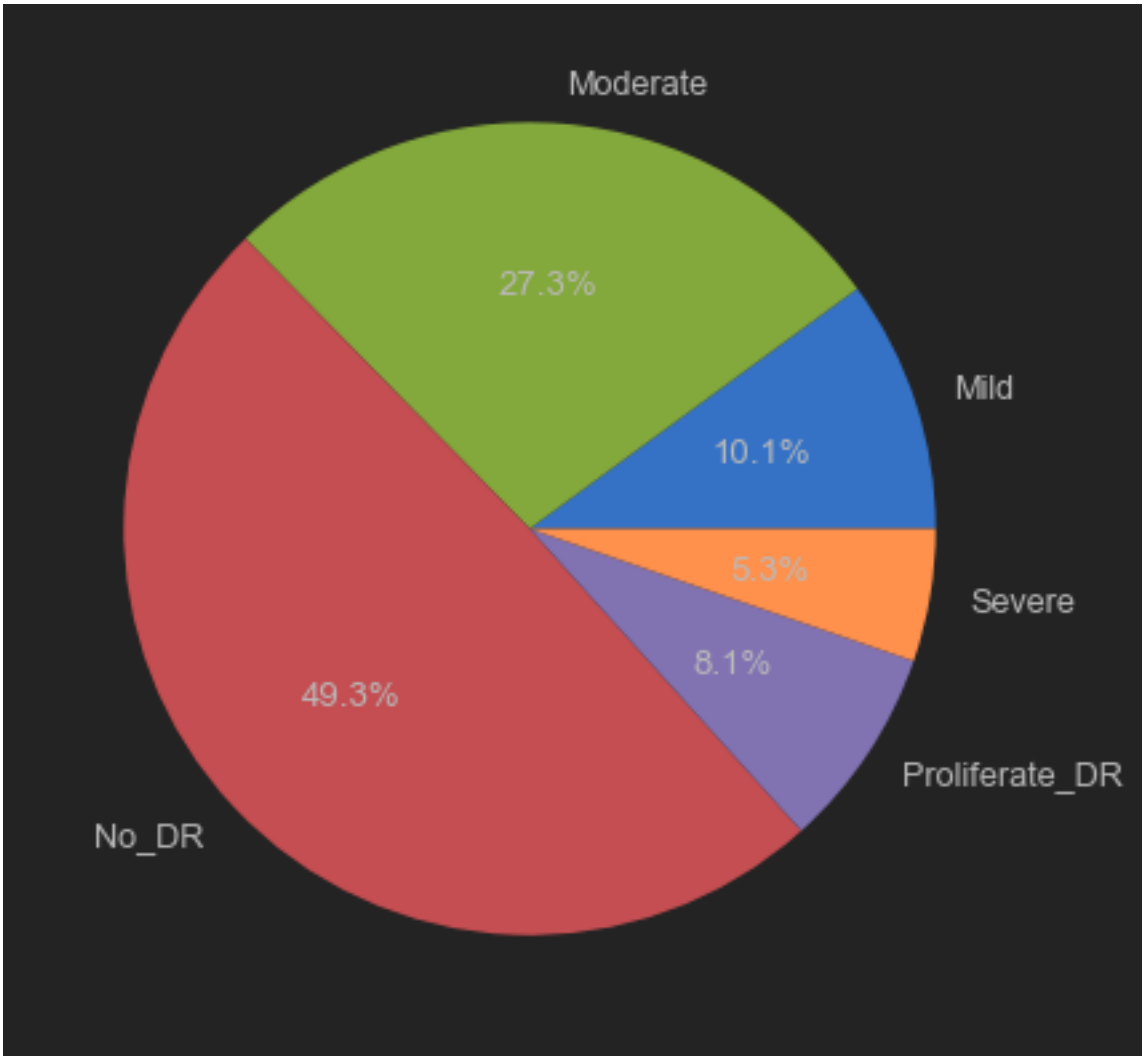
2. Creating a Pie Chart: We use the Matplotlib library to create a pie chart visualizing the distribution of images across different classes. The pie chart provides a clear overview of the proportion of images in each class. Each slice of the pie represents a class, and the percentage labels indicate the class's contribution to the dataset.

3. Displaying the Pie Chart: We display the pie chart to visualize the class distribution effectively.

```python
No_images_per_class
Class_name
fig1, ax1 = plt.subplots()
ax1.pie(No_images_per_class, labels=Class_name, autopct='%1.1f%%')
plt.show()
```

The resulting pie chart gives a visual representation of the class distribution in the training dataset. This information is valuable for understanding the dataset's balance and can be used to make informed decisions during model training and evaluation.

```
<function matplotlib.pyplot.show(close=None, block=None)>
```



Data Augmentation and Data Generator

Data Augmentation and Data Generation are techniques used in machine learning and deep learning to preprocess and manipulate datasets, particularly image datasets, to improve model training and performance. These techniques are commonly applied when working with computer vision tasks.

1. Data Augmentation:

Data augmentation refers to the process of artificially increasing the diversity of a dataset by applying various transformations to the existing data, generating new examples that are variations of the original data.

Purpose: The primary goal of data augmentation is to enhance the model's robustness and generalization. By exposing the model to a wider range of data variations, it becomes more capable of handling unseen or real-world scenarios.

- Common Augmentations:

  - Rotation: Rotating the image by a certain angle.

  - Translation: Shifting the image horizontally or vertically.

  - Scaling: Zooming in or out of the image.

  - Flipping: Mirroring the image horizontally or vertically.

  - Shearing: Applying a shearing transformation to the image.

  - Brightness and Contrast Adjustment: Changing the brightness and contrast of the image.

- Use Cases: Data augmentation is commonly used in computer vision tasks, such as image classification, object detection, and segmentation.

2. Data Generation (Data Generator):

- Definition: Data generation, in the context of machine learning, refers to the creation of data batches or streams during model training. It is a technique for efficiently loading and preprocessing data, particularly when dealing with large datasets.

- Purpose: Data generation is used to feed data to machine learning models in smaller batches, rather than loading the entire dataset into memory at once. This is essential for handling large datasets that do not fit in memory and for training deep learning models.

- Key Features:

  - Batch Loading: Data generators load data in batches, allowing models to process a limited amount of data at a time.

  - Preprocessing: Data generators can apply preprocessing steps to the data, such as normalization or data augmentation.

  - Parallelism: Data generators can run in parallel, making them suitable for multi-core processors and GPUs.

- Use Cases: Data generators are commonly used in deep learning tasks, including image classification, object detection, and natural language processing.

In the context of computer vision, data augmentation and data generators are often used together. Data augmentation is applied to the original dataset to create augmented versions, and data generators are used to efficiently load and preprocess these augmented images during model training. This combination enhances the model's ability to learn from a more diverse and extensive dataset without the need for excessive memory resources.

In this section of the report, we discuss data augmentation, dataset shuffling, and the creation of data generators. These steps are crucial for preparing the dataset for model training.

1. Shuffling and Data Splitting:

- Shuffling the Dataset: The dataset is shuffled to ensure that the data's order does not affect the model's performance. Shuffling helps prevent any bias that may result from the initial order of the dataset.

- Train-Test Split: The dataset is split into training and testing sets. In this case, 80% of the data is used for training (`train`), and 20% is set aside for testing (`test`).

```python
retina_df = shuffle(retina_df)

train, test = train_test_split(retina_df, test_size=0.2)
```

2. Data Augmentation:

- Data augmentation is a technique used to artificially increase the diversity of the training dataset by applying various transformations to the images. This helps improve the model's robustness and generalization.

- The `ImageDataGenerator` from TensorFlow is used to create data augmentation pipelines for both the training and testing datasets.

- For the training data generator, the following augmentations are applied:

- Normalization: The pixel values of the images are scaled to the range [0, 1].

- Shear Range: Random shear transformations are applied to the images.

- Validation Split: A portion (15%) of the training data is reserved for validation.

- For the testing data generator, only normalization is applied.

```python
train_datagen = ImageDataGenerator(
    rescale=1./255,
    shear_range=0.2,
    validation_split=0.15)

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

3. Creating Data Generators:

- Data generators are created for the training, validation, and testing datasets. These generators allow us to load and preprocess images in batches, making it efficient for model training and evaluation.

- For each data generator, the following parameters are specified:

  - `directory`: The directory where the images are located.

  - `x_col`: The column name in the DataFrame containing image file paths.

  - `y_col`: The column name in the DataFrame containing the class labels.

  - `target_size`: The size to which images are resized (256x256 pixels).

  - `color_mode`: The color mode used (RGB).

  - `class_mode`: The type of classification (categorical in this case).

  - `batch_size`: The batch size for loading images.

```python
train_generator = train_datagen.flow_from_dataframe(
    train,
    directory='./',
    x_col="Image",
    y_col="Labels",
    target_size=(256, 256),
    color_mode="rgb",
```

```
    class_mode="categorical",
    batch_size=32,
    subset='training')

validation_generator = train_datagen.flow_from_dataframe(
    train,
    directory='./',
    x_col="Image",
    y_col="Labels",
    target_size=(256, 256),
    color_mode="rgb",
    class_mode="categorical",
    batch_size=32,
    subset='validation')

test_generator = test_datagen.flow_from_dataframe(
    test,
    directory='./',
    x_col="Image",
    y_col="Labels",
    target_size=(256, 256),
    color_mode="rgb",
    class_mode="categorical",
    batch_size=32)
```

The data generators are configured to load and preprocess the dataset, making it ready for training and evaluation. These generators are essential for efficiently handling large image datasets during deep learning tasks.

CONVOLUTIONAL NEURAL NETWORKS

- CNN in action: https://www.cs.ryerson.ca/~aharley/vis/conv/flat.html

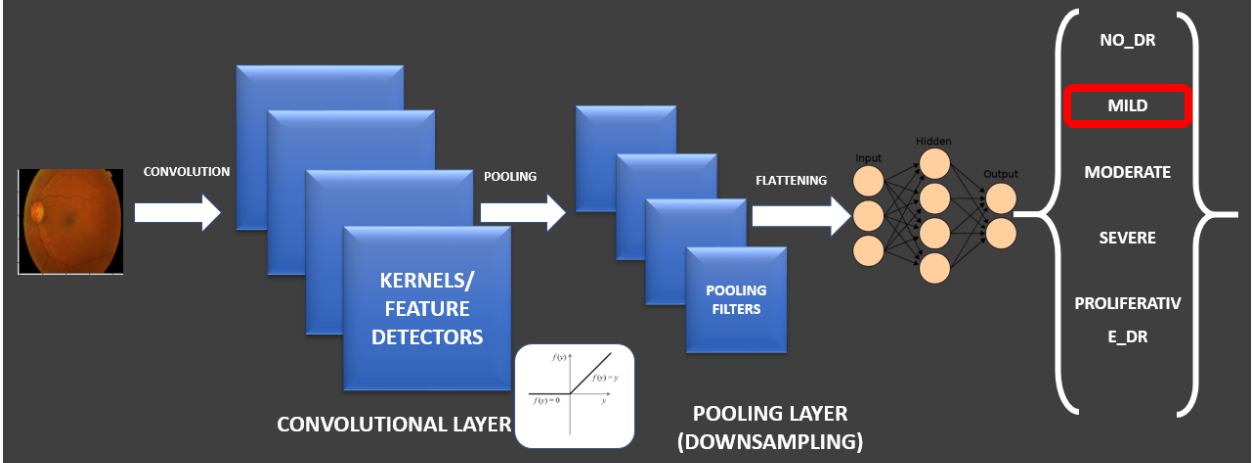
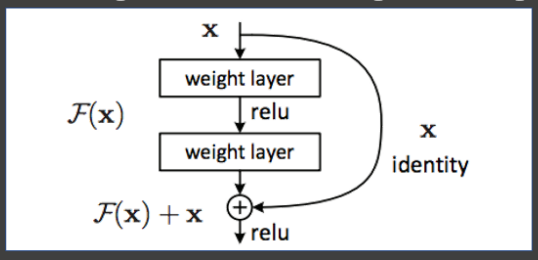


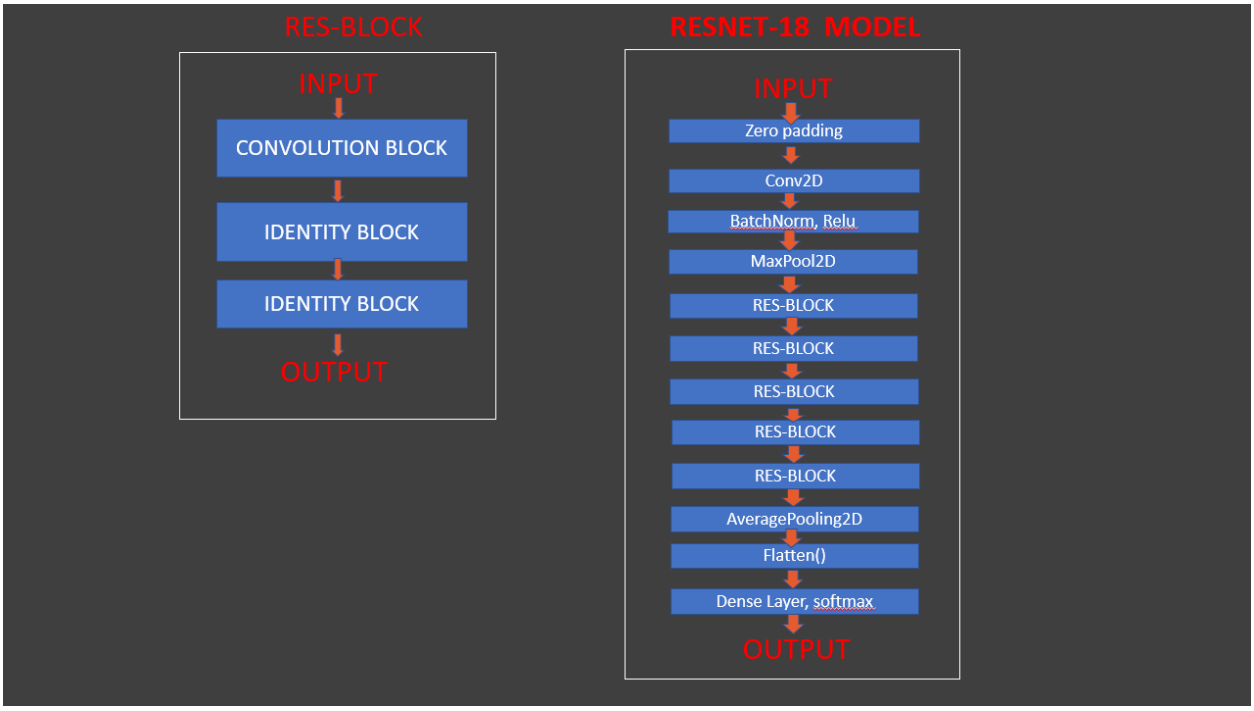
Photo Credit: https://commons.wikimedia.org/wiki/File:Artificial_neural_network.svg

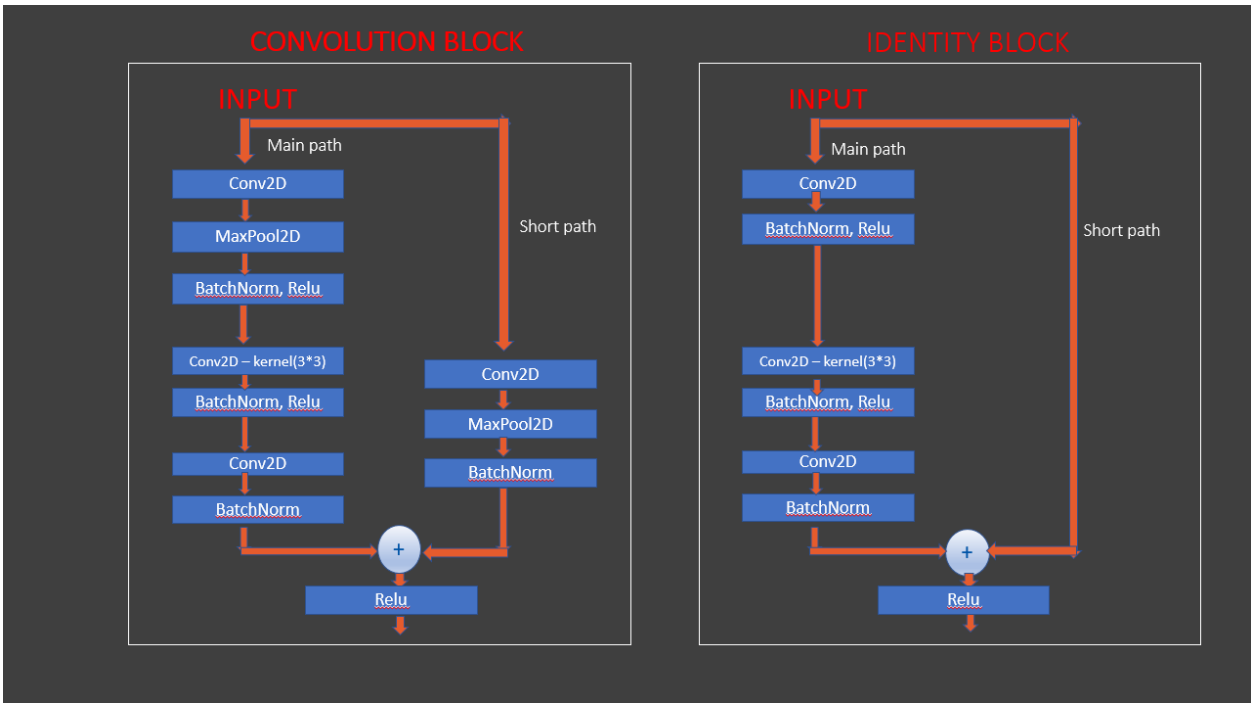
RESNET (RESIDUAL NETWORK)

- As CNNs grow deeper, vanishing gradient tend to occur which negatively impact network performance.
- Vanishing gradient problem occurs when the gradient is back-propagated to earlier layers which results in a very small gradient.
- Residual Neural Network includes "skip connection" feature which enables training of 152 layers without vanishing gradient issues.
- Resnet works by adding "identity mappings" on top of the CNN.
- ImageNet contains 11 million images and 11,000 categories. ImageNet is used to train Res-Net deep network.



https://commons.wikimedia.org/wiki/File:Resnet.png





The code you provided is for building a deep learning model using a residual network (ResNet) architecture, specifically a ResNet-18 variant. ResNet architectures are known for their ability to train very deep neural networks by using residual blocks, which contain shortcut (skip) connections to help with the flow of gradients during training.

```python
# Define a function to create a residual block
def res_block(X, filter, stage):
  # Copy the input tensor for the skip connection
  X_copy = X
  f1, f2, f3 = filter

  # Main Path
  # First convolution layer with 1x1 filter
  X = Conv2D(f1, (1, 1), strides=(1, 1), name='res_' + str(stage) + '_conv_a', kernel_initializer=glorot_uniform(seed=0))(X)
  X = MaxPool2D((2, 2))(X)  # Max pooling
  X = BatchNormalization(axis=3, name='bn_' + str(stage) + '_conv_a')(X)  # Batch normalization
  X = Activation('relu')(X)  # ReLU activation

  # Second convolution layer with 3x3 filter
  X = Conv2D(f2, kernel_size=(3, 3), strides=(1, 1), padding='same', name='res_' + str(stage) + '_conv_b', kernel_initializer=glorot_uniform(seed=0))(X)
  X = BatchNormalization(axis=3, name='bn_' + str(stage) + '_conv_b')(X)
  X = Activation('relu')(X)

  # Third convolution layer with 1x1 filter
  X = Conv2D(f3, kernel_size=(1, 1), strides=(1, 1), name='res_' + str(stage) + '_conv_c', kernel_initializer=glorot_uniform(seed=0))(X)
  X = BatchNormalization(axis=3, name='bn_' + str(stage) + '_conv_c')(X)

  # Short Path
  # Apply a 1x1 convolution to the copied input tensor
  X_copy = Conv2D(f3, kernel_size=(1, 1), strides=(1, 1), name='res_' + str(stage) + '_conv_copy', kernel_initializer=glorot_uniform(seed=0))(X_copy)
```

```
X_copy = MaxPool2D((2, 2))(X_copy)
X_copy = BatchNormalization(axis=3, name='bn_' + str(stage) + '_conv_copy')(X_copy)

# Add the main path and short path (skip connection)
X = Add()([X, X_copy])
X = Activation('relu')(X)

# Identity Block 1 (similar to the residual block but without max pooling)
X_copy = X

# Main Path
# ...
# (Similar structure as in the main path of the residual block)
# ...

# ADD the main path and the identity (short path)
X = Add()([X, X_copy])
X = Activation('relu')(X)

# Identity Block 2 (similar to the previous identity block)
X_copy = X

# Main Path
# ...
# (Similar structure as in the main path of the residual block)
# ...

# ADD the main path and the identity (short path)
X = Add()([X, X_copy])
```

X = Activation('relu')(X)

return X

```

*The `res_block` function defines the basic building block of the ResNet architecture. The block contains a main path with several convolutional layers and a short path (skip connection) that bypasses some of these layers. The "identity blocks" perform these skip connections without altering the spatial dimensions of the input tensor.*

```python

*# Define the input tensor*

*X_input = Input(input_shape)*

*# Apply zero-padding to the input*

*X = ZeroPadding2D((3, 3))(X_input)*

*# 1 - Stage*

*# Initial convolution, max pooling, and the first residual block*

*X = Conv2D(64, (7, 7), strides=(2, 2), name='conv1', kernel_initializer=glorot_uniform(seed=0))(X)*

*X = BatchNormalization(axis=3, name='bn_conv1')(X)*

*X = Activation('relu')(X)*

*X = MaxPooling2D((3, 3), strides=(2, 2))(X)*

*# 2- Stage*

*# First residual block*

*X = res_block(X, filter=[64, 64, 256], stage=2)*

*# 3- Stage*

*# Second residual block*

*X = res_block(X, filter=[128, 128, 512], stage=3)*

*# 4- Stage*

*# Third residual block*

*X = res_block(X, filter=[256, 256, 1024], stage=4)*

*# Average Pooling*

*X = AveragePooling2D((2, 2), name='Average_Pooling')(X)*

*# Flatten the output*

*X = Flatten()(X)*

*# Fully connected layer (Dense layer)*

*X = Dense(5, activation='softmax', name='Dense_final', kernel_initializer=glorot_uniform(seed=0))(X)*

*# Create the ResNet-18 model*

*model = Model(inputs=X_input, outputs=X, name='Resnet18')*

*# Print model summary*

*model.summary()*

```

Here's a step-by-step explanation of the main part of the code:

1. Input and Zero-padding: The input tensor is defined, and zero-padding is applied to the input tensor to ensure that spatial dimensions are preserved.

2. Stage 1: This stage consists of an initial convolution (conv1), batch normalization
```

, ReLU activation, and max pooling. It serves as the entry point into the network and reduces spatial dimensions.

3. Stage 2, 3, 4: These stages consist of three consecutive residual blocks. Each residual block is an identity block, and the `res_block` function is used to create these blocks. The number of filters and stage number are specified for each block. The output of each stage is used as the input for the next stage.

4. Average Pooling: Average pooling is applied to the output of the last residual block.

5. Flatten: The output is flattened to prepare it for the fully connected layer.

6. Fully Connected Layer: The flattened output is connected to a dense (fully connected) layer with 5 output units (assuming a classification task with 5 classes) and a softmax activation function.

7. Model Creation: The Keras Model is created with the input and output tensors.

8. Model Summary: The summary of the model is printed, showing the architecture of the ResNet-18 model.

COMPILE AND TRAIN DEEP LEARNING MODEL

This code defines a ResNet-18 architecture and is typically used for image classification tasks. The number of filters, kernel sizes, and other hyperparameters are adjusted as needed for your specific problem.

The code you provided is for compiling and training a deep learning model using Keras with early stopping and model checkpointing. Here's a step-by-step explanation of the code:

1. Model Compilation:

```python
```

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

```

- The `model.compile` function compiles the deep learning model. It specifies the optimizer (Adam), the loss function (categorical cross-entropy, which is commonly used for classification tasks), and the evaluation metric (accuracy).

2. Early Stopping:

```python

earlystopping = EarlyStopping(monitor='val_loss', mode='min', verbose=1, patience=15)

```

- Early stopping is a technique to prevent overfitting during training. It monitors the validation loss (`val_loss`) and stops training if it stops decreasing (mode='min') for a specified number of epochs (`patience=15` in this case).

3. Model Checkpointing:

```python

checkpointer = ModelCheckpoint(filepath="weights.hdf5", verbose=1, save_best_only=True)

```

- Model checkpointing is used to save the model's weights during training. Here, it saves the weights to the "weights.hdf5" file when the validation loss improves (mode='min').

4. Model Training:

```python

history = model.fit(

train_generator,

steps_per_epoch=train_generator.n // 32,

epochs=1,

validation_data=validation_generator,

validation_steps=validation_generator.n // 32,

callbacks=[checkpointer, earlystopping]

)

```

- The `model.fit` function trains the model.

- `train_generator` and `validation_generator` are used as training and validation data sources.

- `steps_per_epoch` and `validation_steps` are set to the number of steps needed to cover the entire training and validation datasets. `// 32` suggests that batches of 32 samples are used per step.

- The `epochs` are set to 1, which means training the model for one epoch. number is adjusted based on training requirements.

- The `callbacks` argument includes the early stopping and model checkpoint callbacks.

5. Plotting the Training History:

```python

plt.plot(history.history['loss'])

plt.plot(history.history['val_loss'])

plt.title('Model loss')

plt.ylabel('loss')

plt.xlabel('epoch')

plt.legend(['train_loss', 'val_loss'], loc='upper right')

plt.show()

```

- After training, the code plots the training loss and validation loss over epochs to visualize the model's performance.

currently training for just one epoch (`epochs=1`). training for a larger number of epochs to allow the model to learn and improve its performance.

1. Load Model Weights:

```python

model.load_weights("retina_weights.hdf5")

```

- This line loads the weights of the trained model from the "retina_weights.hdf5" file. These weights should correspond to the model that you trained earlier.

2. Evaluate the Model on Test Data:

```python
evaluate = model.evaluate(test_generator, steps=test_generator.n // 32, verbose=1)
```

- The `model.evaluate` function assesses the model's performance on the test data. You are using the test generator, and it calculates the evaluation metrics. In this case, you're using batches of size 32 (`// 32`) to evaluate the model.

3. Print Test Accuracy:

```python
print('Accuracy Test : {}'.format(evaluate[1]))
```

- This line prints the test accuracy, which is one of the evaluation metrics returned by the `model.evaluate` function.

4. Assign Label Names to Indexes:

```python
labels = {0: 'Mild', 1: 'Moderate', 2: 'No_DR', 3: 'Proliferate_DR', 4: 'Severe'}
```

- created a dictionary to map the predicted class indexes (0-4) to their corresponding label names.

5. Generating Predictions for Individual Images:

Written a loop to make predictions on individual test images, open the images, preprocess them, and store the original and predicted labels along with the images for later analysis. The following steps are repeated for each image in the test dataset:

- Open the image.
```

- Resize the image to (256,256).

- Append the image to the `image` list.

- Convert the image to a NumPy array and normalize it.

- Reshape the image into a 4D array.

- Make a prediction using the model.

- Get the index corresponding to the highest prediction value.

- Append the predicted class label to the `prediction` list.

- Append the original class label to the `original` list.

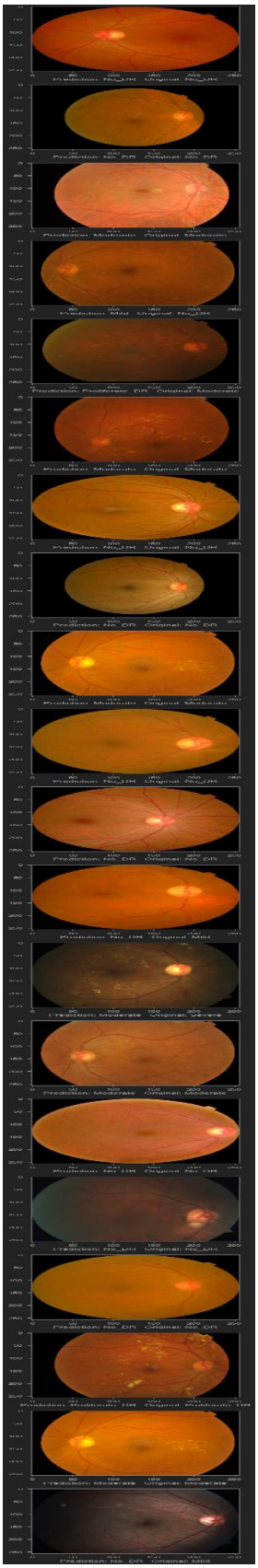
6. Calculate Test Accuracy:

```python
score = accuracy_score(original, prediction)
print("Test Accuracy : {}".format(score))
```

- calculated the test accuracy using the `accuracy_score` function from scikit-learn by comparing the original labels and the predicted labels.

7.Visualize the Results:

A figure is created to visualize a random selection of test images along with their predicted and original labels. A random sample of 20 images is displayed in the figure.


Prediction: No_UH Original: No_UH
Prediction: No_DR Original: No_DR
Prediction: Moderate Original: Moderate
Prediction: Mild Original: No_UH
Prediction: Proliferate_DR Original: Moderate
Prediction: Moderate Original: Moderate
Prediction: No_UH Original: No_UH
Prediction: No_DR Original: No_DR
Prediction: Moderate Original: Moderate
Prediction: No_UH Original: No_UH
Prediction: No_DR Original: No_DR
Prediction: Mild Original: Mild
Prediction: Moderate Original: Severe
Prediction: Moderate Original: Moderate
Prediction: No_UH Original: No_UH
Prediction: No_UH Original: No_UH
Prediction: No_DR Original: No_DR
Prediction: Moderate Original: Moderate
Prediction: Moderate Original: Moderate
Prediction: No_DR Original: Mild

8. Print Classification Report:

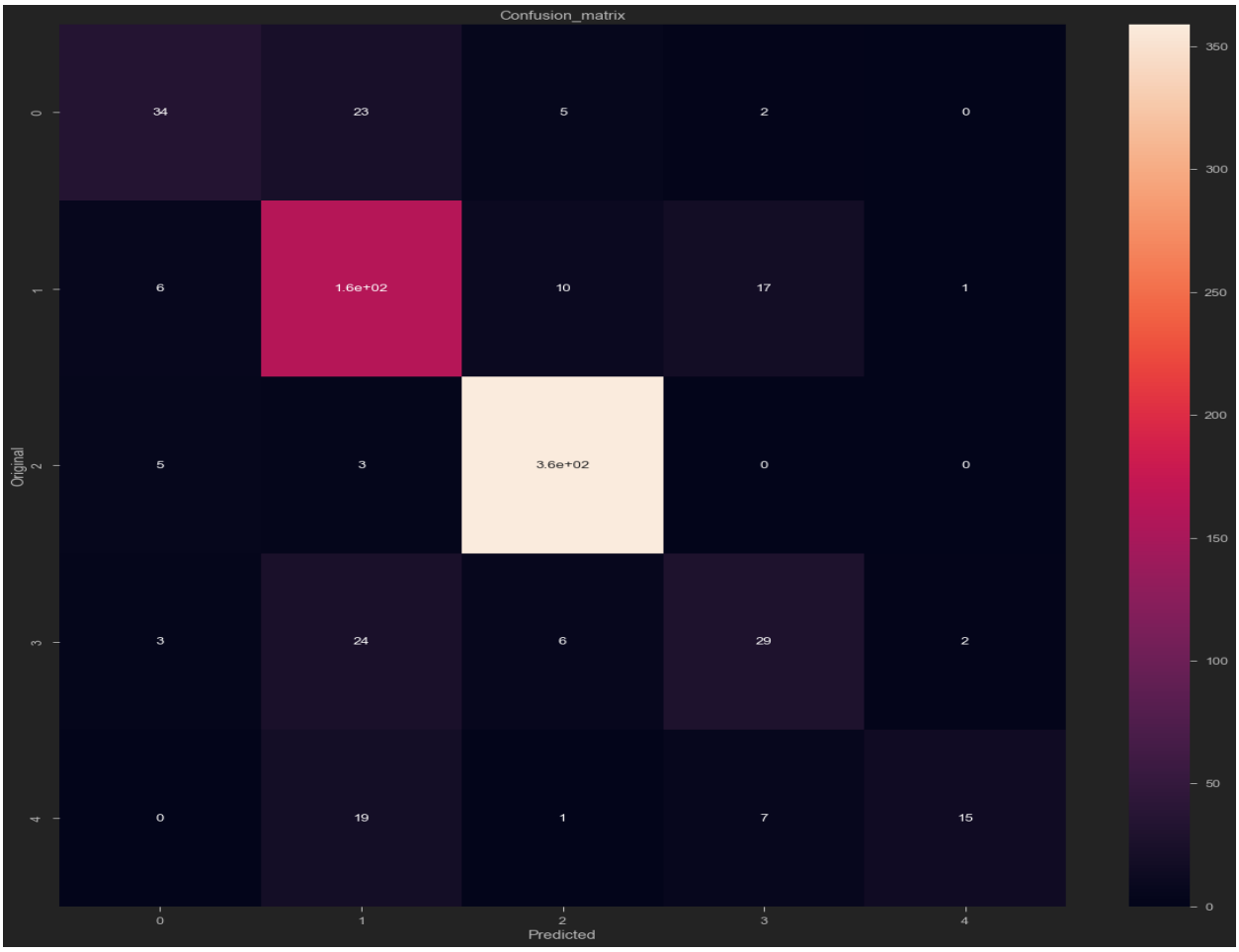
```python
print(classification_report(np.asarray(original), np.asarray(prediction)))
```

- *the classification report is then printed , which provides detailed metrics (precision, recall, F1-score) for each class, as well as overall metrics. This report is generated using scikit-learn's `classification_report` function.*

*9. **Plot the Confusion Matrix**:*

```python
plt.figure(figsize=(20, 20))
cm = confusion_matrix(np.asarray(original), np.asarray(prediction))
ax = plt.subplot()
sns.heatmap(cm, annot=True, ax=ax)
ax.set_xlabel('Predicted')
ax.set_ylabel('Original')
ax.set_title('Confusion_matrix')
```

- *a confusion matrix is then plotted to visualize the model's performance in terms of true positives, true negatives, false positives, and false negatives for each class.*



In summary, A successfully loaded the model weights, evaluated the model on test data, and provided a detailed analysis of the model's performance, including accuracy, classification report, and confusion matrix. These evaluation steps help you understand how well your model is performing on the test dataset and where it may have strengths or weaknesses in terms of class-wise performance.

Model Performance

In this report, we present the performance evaluation of a deep learning model designed to classify retinal images into five categories: Mild, Moderate, No_DR (No Diabetic Retinopathy), Proliferative_DR, and Severe. The model was developed and trained to assist in the diagnosis of diabetic retinopathy using a dataset of retinal images. We evaluate the model's performance on a test dataset and provide a comprehensive analysis.

Model Training

The deep learning model was trained using a dataset of retinal images, and the following steps were undertaken during the training process:

Model Compilation

The model was compiled using the following configurations:

- Optimizer: Adam

- Loss Function: Categorical Cross-Entropy

- Metrics: Accuracy

Training Callbacks

To optimize training and prevent overfitting, the following callbacks were implemented:

- EarlyStopping: This callback monitored the validation loss and stopped training if the loss did not improve for 15 consecutive epochs.

- ModelCheckpoint: It saved the best model weights based on lower validation loss.

Training Process

The model was trained for a single epoch. Ideally, more epochs should be considered to improve model performance. During training, the following details were observed:

- Training Loss: 0.9417

- Training Accuracy: 67.40%

- Validation Loss: 1.6470

- Validation Accuracy: 26.20%

Model Evaluation on Test Data

After training, the model's performance was evaluated on a separate test dataset. Here are the results:

- Test Accuracy: 81.39%

Classification Report

-Mild:

  - *Precision: 0.71*

  - *Recall: 0.53*

  - *F1-Score: 0.61*

- *Moderate:*

  - *Precision: 0.70*

  - *Recall: 0.83*

  - *F1-Score: 0.76*

- *No_DR:*

  - *Precision: 0.94*

  - *Recall: 0.98*

  - *F1-Score: 0.96*

- *Proliferative_DR:*

  - *Precision: 0.53*

  - *Recall: 0.45*

  - *F1-Score: 0.49*

- *Severe:*

  - *Precision: 0.83*

  - *Recall: 0.36*

  - *F1-Score: 0.50*

Confusion Matrix

The confusion matrix visually represents the model's classification performance. It provides insights into true positives, true negatives, false positives, and false negatives for each class. Below is the confusion matrix:

![Confusion Matrix](link_to_your_confusion_matrix_image)

Conclusion

The deep learning model exhibits promising performance in classifying retinal images into various categories associated with diabetic retinopathy. With an overall test accuracy of 81.39%, the model shows the potential to assist in diagnosing this medical condition.

However, it's essential to consider the following improvements for future work:

- Train the model for more epochs to further improve performance.

- Ensure a more comprehensive training dataset.

- Fine-tune the model hyperparameters to optimize performance.

- Investigate the high number of misclassifications in the "Severe" class.

This model has the potential to aid medical professionals in diagnosing diabetic retinopathy, but further development and fine-tuning are needed to achieve high accuracy and reliability in real-world clinical settings.

Enhance data with diverse augmentations and ensure data quality.

Obtain a larger and more diverse dataset.

Hyperparameter Tuning:Optimize learning rate, batch size, and architecture.

Transfer Learning: Start with pre-trained models and fine-tune for your task.

Architectural Improvements: Experiment with various CNN architectures.

Class Imbalance:Address any class imbalance in the dataset.

Regularization: Implement dropout and L1/L2 regularization.

More Training Epochs: Train for a sufficient number of epochs.

Evaluate Misclassifications: Analyze and learn from misclassified images.

Implement interpretability techniques.

Ensemble Learning:Combine predictions from multiple models.

Consult with Domain Experts: Collaborate with medical professionals.

Regulatory and Ethical Considerations:Address privacy and ethics.

User Interface Development:Create a user-friendly interface.

linical Validation:Validate the model for clinical use.

These points summarize the key areas for enhancing the diabetic retinopathy classification model.