Below is a line-by-line explanation of the provided code, with each segment explained in a simple, beginner-friendly way, focusing on **what is happening**, **why it works like that**, and additional details about each concept.

**Importing Libraries**

import numpy as np

import matplotlib.pyplot as plt

import tensorflow as tf

from tensorflow.keras import layers, Sequential, Input

1. **numpy**:
   * Used for numerical operations, such as working with arrays and manipulating data.
   * Often used to handle model predictions and prepare image data.
2. **matplotlib.pyplot**:
   * Library for creating plots and visualizations.
   * Helps display the training progress (e.g., accuracy, loss) and visualize sample images.
3. **tensorflow**:
   * A deep learning framework for building and training neural networks.
   * It provides tools to process datasets and design machine learning models.
4. **from tensorflow.keras**:
   * **layers**: Used to define the individual building blocks (layers) of the neural network.
   * **Sequential**: A simple way to stack layers linearly for building a neural network.
   * **Input**: Specifies the input shape of the data entering the model.

**Paths to Datasets**

train\_path = 'train'

validation\_path = 'validation'

test\_path = 'test'

* The datasets are organized in directories:
  + train: Images for training the model.
  + validation: Images for validating the model during training.
  + test: Images for evaluating the model's final performance.
* **Why separate datasets?**
  + **Train**: Teaches the model to recognize patterns.
  + **Validation**: Monitors the model during training to check if it is learning effectively.
  + **Test**: Measures the model's performance on unseen data.

**Define Image Dimensions and Batch Size**

img\_height = 180

img\_width = 180

batch\_size = 32

* **Image dimensions**:
  + Input images are resized to 180x180 pixels to ensure consistent size for the model.
  + Deep learning models require fixed input sizes.
* **Batch size**:
  + The number of images processed in one step during training.
  + A batch size of 32 means the model processes 32 images at a time.

**Loading the Datasets**

train\_ds = tf.keras.utils.image\_dataset\_from\_directory(

train\_path,

image\_size=(img\_height, img\_width),

batch\_size=batch\_size,

shuffle=True

)

* **image\_dataset\_from\_directory**:
  + Loads images from the train\_path directory and organizes them into batches.
  + Resizes all images to 180x180.
  + **shuffle=True**: Randomizes the order of images to prevent bias during training.

**Validation and Test Datasets**

val\_ds = tf.keras.utils.image\_dataset\_from\_directory(

validation\_path,

image\_size=(img\_height, img\_width),

batch\_size=batch\_size,

shuffle=False

)

test\_ds = tf.keras.utils.image\_dataset\_from\_directory(

test\_path,

image\_size=(img\_height, img\_width),

batch\_size=batch\_size,

shuffle=False

)

* The **validation** and **test datasets** are loaded similarly, but:
  + **shuffle=False**: Ensures the image order is preserved for consistent evaluation.

**Class Names**

class\_names = train\_ds.class\_names

print("Detected classes:", class\_names)

* Extracts the class labels (e.g., Healthy, Mild DR) from the dataset directories.
* This allows the model to map predictions to human-readable labels.

**Visualizing Sample Images**

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

for images, labels in train\_ds.take(1):

for i in range(5):

plt.subplot(3, 3, i + 1)

plt.imshow(images[i].numpy().astype('uint8'))

plt.title(class\_names[labels[i]])

plt.axis("off")

* **train\_ds.take(1)**: Fetches one batch of 32 images.
* **images[i]**: Accesses individual images from the batch.
* **plt.imshow()**: Displays the image.
* **plt.title()**: Adds the class name as the title.
* **Why visualize?**
  + To ensure the dataset is loaded correctly and verify labels.

**Dataset Optimization**

AUTOTUNE = tf.data.AUTOTUNE

train\_ds = train\_ds.prefetch(buffer\_size=AUTOTUNE)

val\_ds = val\_ds.prefetch(buffer\_size=AUTOTUNE)

test\_ds = test\_ds.prefetch(buffer\_size=AUTOTUNE)

* **AUTOTUNE**:
  + Automatically adjusts the data loading process for maximum performance.
  + Prefetching overlaps data loading with model computation, reducing idle time.
  + This speeds up training.

**Building the Model**

model = Sequential([

Input(shape=(img\_height, img\_width, 3)),

layers.Rescaling(1./255),

layers.Conv2D(32, 3, activation='relu'),

layers.MaxPooling2D(),

layers.Conv2D(64, 3, activation='relu'),

layers.MaxPooling2D(),

layers.Conv2D(128, 3, activation='relu'),

layers.MaxPooling2D(),

layers.Flatten(),

layers.Dropout(0.5),

layers.Dense(128, activation='relu'),

layers.Dense(len(class\_names), activation='softmax')

])

* **Why Sequential?**
  + Layers are stacked in order, making it simple to build.
  + Works well for basic image classification problems.

**Layers:**

1. **Input**:
   * Specifies the shape of the input image: 180x180x3 (width, height, RGB channels).
2. **Rescaling**:
   * Normalizes pixel values to the range [0, 1] for better performance.
3. **Conv2D**:
   * Extracts features from the image (e.g., edges, textures).
   * **32, 64, 128**: Number of filters that detect patterns of increasing complexity.
4. **MaxPooling2D**:
   * Reduces the size of feature maps, retaining the most important features.
5. **Flatten**:
   * Converts the 2D feature maps into a 1D vector for the dense layers.
6. **Dropout**:
   * Prevents overfitting by randomly turning off 50% of neurons during training.
7. **Dense**:
   * Fully connected layers for classification.
   * Final layer has len(class\_names) outputs with softmax to produce probabilities.

**Compiling the Model**

model.compile(optimizer='adam',

loss='sparse\_categorical\_crossentropy',

metrics=['accuracy'])

* **adam**:
  + An adaptive optimizer for faster convergence.
* **sparse\_categorical\_crossentropy**:
  + Loss function for multi-class classification.
  + Suitable when labels are integers (e.g., 0, 1, 2).
* **metrics=['accuracy']**:
  + Tracks how many predictions match the true labels.

**Training the Model**

history = model.fit(train\_ds, validation\_data=val\_ds, epochs=15)

* **fit()**:
  + Trains the model for 15 epochs (complete passes through the training data).
  + Uses val\_ds to validate the model's performance after each epoch.

**Visualizing Training Progress**

plt.figure(figsize=(12, 6))

plt.subplot(1, 2, 1)

plt.plot(range(epochs), history.history['accuracy'], label='Training Accuracy')

plt.plot(range(epochs), history.history['val\_accuracy'], label='Validation Accuracy')

plt.legend()

plt.title('Training and Validation Accuracy')

plt.subplot(1, 2, 2)

plt.plot(range(epochs), history.history['loss'], label='Training Loss')

plt.plot(range(epochs), history.history['val\_loss'], label='Validation Loss')

plt.legend()

plt.title('Training and Validation Loss')

* Plots show:
  + Accuracy increasing over epochs (indicating learning).
  + Loss decreasing (indicating improved predictions).

**Testing the Model**

test\_loss, test\_accuracy = model.evaluate(test\_ds)

print(f"Test Accuracy: {test\_accuracy:.2f}")

* Evaluates the model on the test dataset, providing final accuracy on unseen data.

**Save and Predict**

model.save('DR\_Detection\_Model.keras')

def predict\_image(image\_path):

img = tf.keras.utils.load\_img(image\_path, target\_size=(img\_height, img\_width))

img\_array = tf.keras.utils.img\_to\_array(img)

img\_array = tf.expand\_dims(img\_array, 0)

predictions = model.predict(img\_array)

score = tf.nn.softmax(predictions[0])

print(f"This image likely belongs to {class\_names[np.argmax(score)]} with a confidence of {100 \* np.max(score):.2f}%.")

* **save()**:
  + Saves the trained model for future use.
* **predict\_image()**:
  + Loads a new image, preprocesses it, and uses the model to predict the class.
  + Outputs the predicted class and confidence.

This step-by-step breakdown ensures a clear understanding of each segment, even for beginners.