12/23/2023

Data Science Tools

Image Classification

|  |  |
| --- | --- |
| Name | ID |
| Ayatullah Nabil Elsayed | **20221452375** |
| Menna Allah Yasser Nabil | **20221451437** |
| Shahd Gamal Saad Zaghlol | **20221452171** |
| Amany Gaber Okaila Khalifa | **20221451708** |
| Marwan Ali Abd-Elsatar | **20221460240** |
| Fouad Essam William | **20221450596** |
| Shahd Khalaf Ahmed Omar | **20221462492** |
| Omar Mohamed Mostafa Mohamed | **2022446471** |
| Zyad Mohamed Abd Elmoniem | **20221453856** |

Table of Contents

[Introduction 2](#_Toc154230270)

[Objectives 2](#_Toc154230271)

[Install independents and setup 3](#_Toc154230272)

[Remove Dodgy images 4](#_Toc154230273)

[Load the dataset 4](#_Toc154230274)

[Scale Data 5](#_Toc154230275)

[Split Data 6](#_Toc154230276)

[Build Deep Learning Model (CNN) 7](#_Toc154230277)

[Train the Model 9](#_Toc154230278)

[Plot Performance 10](#_Toc154230279)

[Evaluate 11](#_Toc154230280)

[Test 12](#_Toc154230281)

[Building deep learning model 14](#_Toc154230282)

[Alexnet 16](#_Toc154230283)

# Introduction

In today's data-driven world, images are a ubiquitous form of information. From personal photos to social media posts, scientific datasets to satellite imagery, understanding the content of images has become crucial in various fields.

This project delves into the fascinating domain of **image classification** using **deep learning convolutional neural networks (CNNs)**. We aim to leverage the power of CNNs, a highly advanced form of artificial intelligence, to automatically analyze and categorize images based on their content.

Imagine a system that can identify objects in photographs, differentiate between emotions in facial expressions, or even diagnose diseases from medical scans. The possibilities are endless, and image classification with CNNs offers a powerful tool to unlock these possibilities.

**Specifically, this project will focus on:**

* **Developing a deep learning CNN model:** We will design and train a CNN architecture tailored to a specific image classification task (e.g., identifying dog breeds, classifying medical images, etc.).
* **Preprocessing and data augmentation:** We will explore techniques to prepare training data for optimal model performance, including resizing, normalization, and augmenting datasets to improve robustness.
* **Training and optimization:** We will train the CNN model on a labeled image dataset, using techniques like backpropagation and gradient descent to adjust its parameters and improve its classification accuracy.
* **Evaluation and analysis:** We will evaluate the trained model's performance on unseen data, analyzing its strengths and weaknesses to identify areas for improvement.

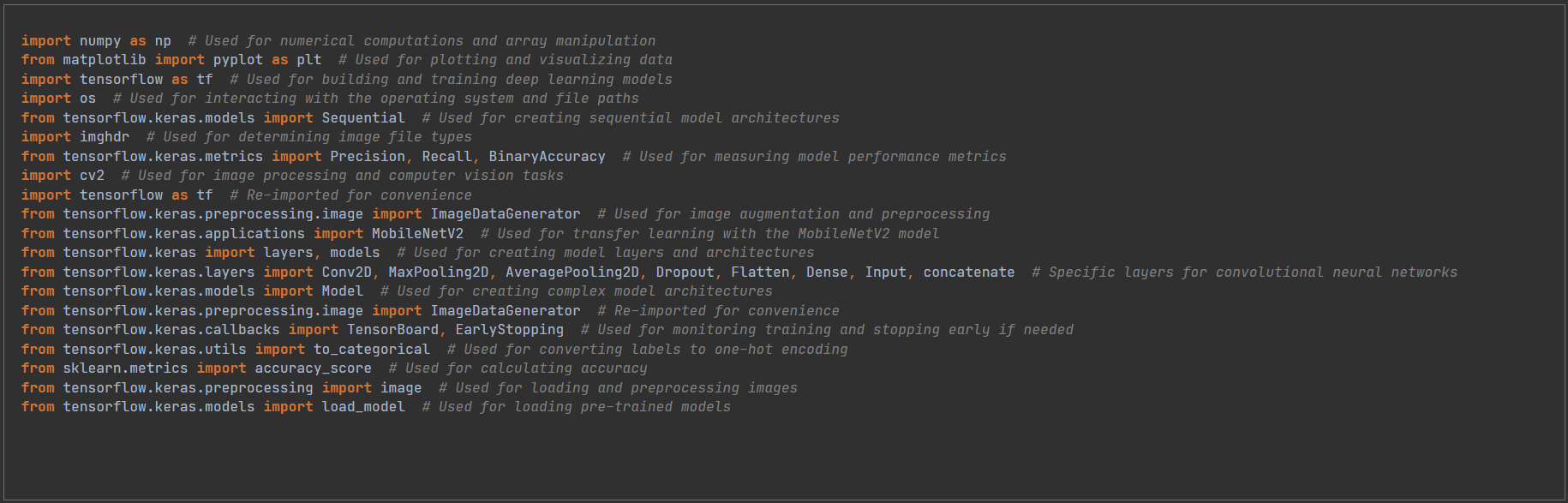
# Objectives

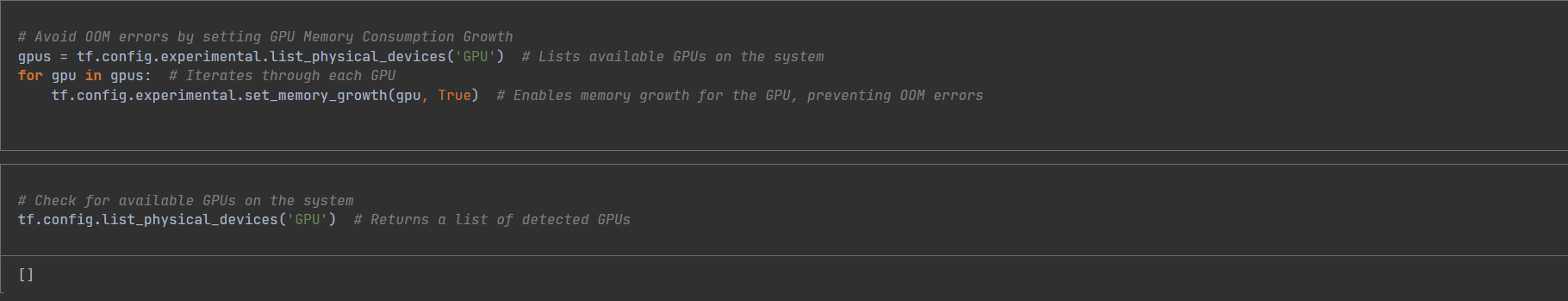
**This project promises to be:**

* **Challenging and rewarding:** Deep learning presents a complex and ever-evolving field, pushing the boundaries of our understanding of artificial intelligence. Successfully implementing a CNN for image classification requires dedication, research, and problem-solving skills.
* **Practical and impactful:** The ability to automatically classify images has real-world applications across diverse domains, from self-driving cars to medical diagnosis to e-commerce.
* **Educational and enlightening:** Through this project, we will gain a deeper understanding of CNN architecture, training methodologies, and the power of deep learning for image analysis.

# Install independents and setup

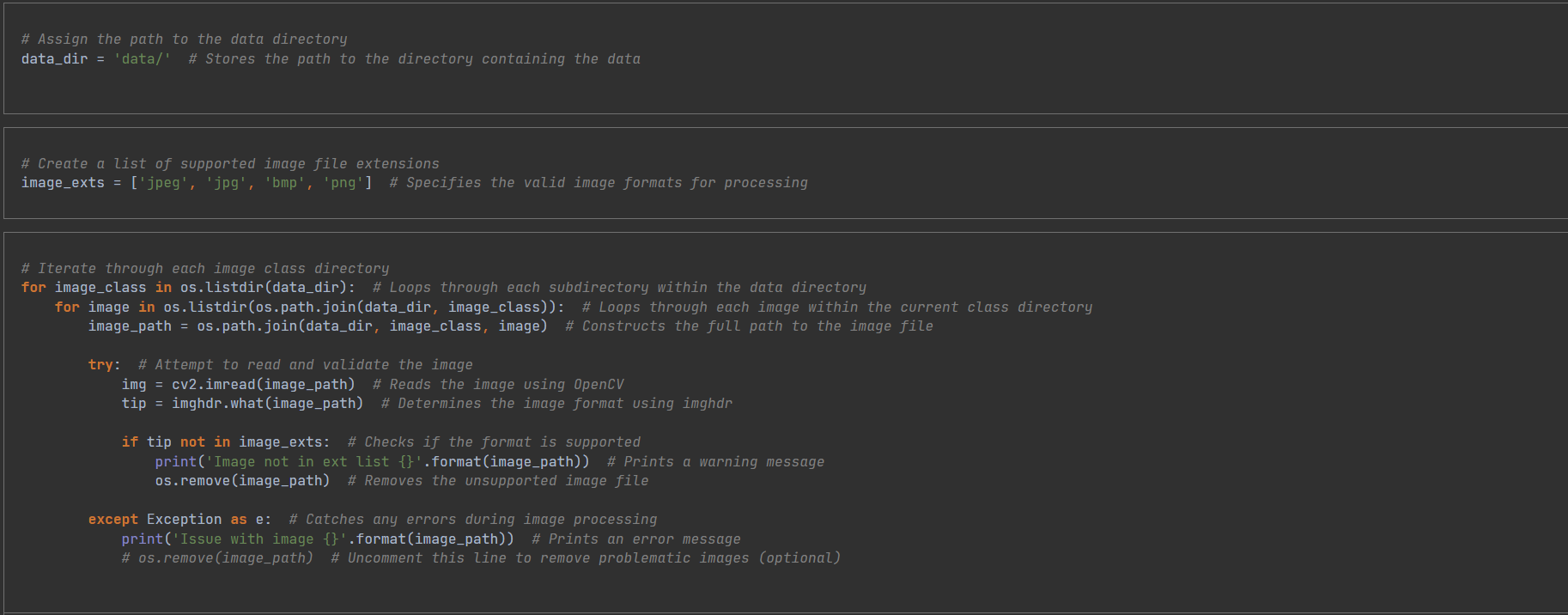
import all required packages with their independents





Efficient GPU memory management is crucial for smooth image classification with deep learning models. This code snippet tackles a common challenge – “Out of Memory” errors during training. By first identifying available GPUs, it then activates “memory growth” on each one. This dynamic allocation avoids pre-allocating all memory at once, a leading cause of OOMs. This optimization allows our image classifier models to train effectively and utilize the full potential of the available GPU resources, paving the way for successful image classification tasks.

# Remove Dodgy images



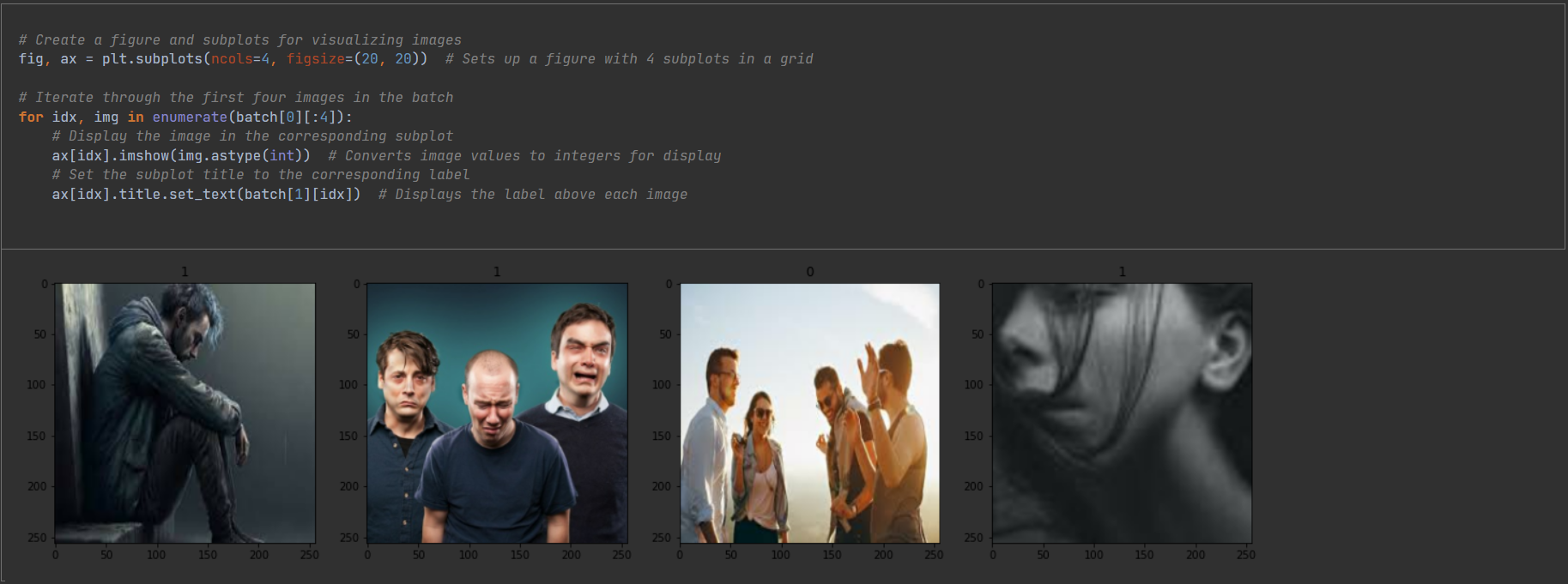
This code snippet ensures data integrity and compatibility for image classification by meticulously inspecting and validating images within the dataset. It first navigates through the organized data directory, which houses images within subdirectories representing their respective classes. It then meticulously examines each image file:

* It validates the image format using OpenCV and imghdr to ensure consistency and compatibility with subsequent processing steps.
* It identifies and removes any unsupported image formats, preventing potential errors during training or inference.
* It diligently handles any exceptions that might arise during image loading or processing, flagging problematic files for further inspection or removal.

This thorough validation process safeguards the quality and consistency of the image dataset, laying a solid foundation for accurate and reliable image classification.

# Load the dataset

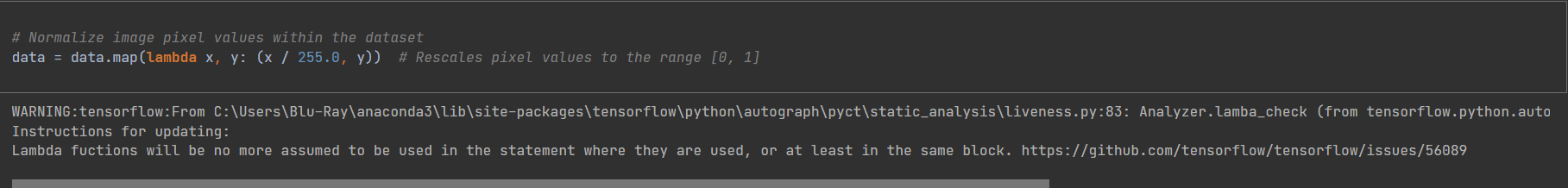




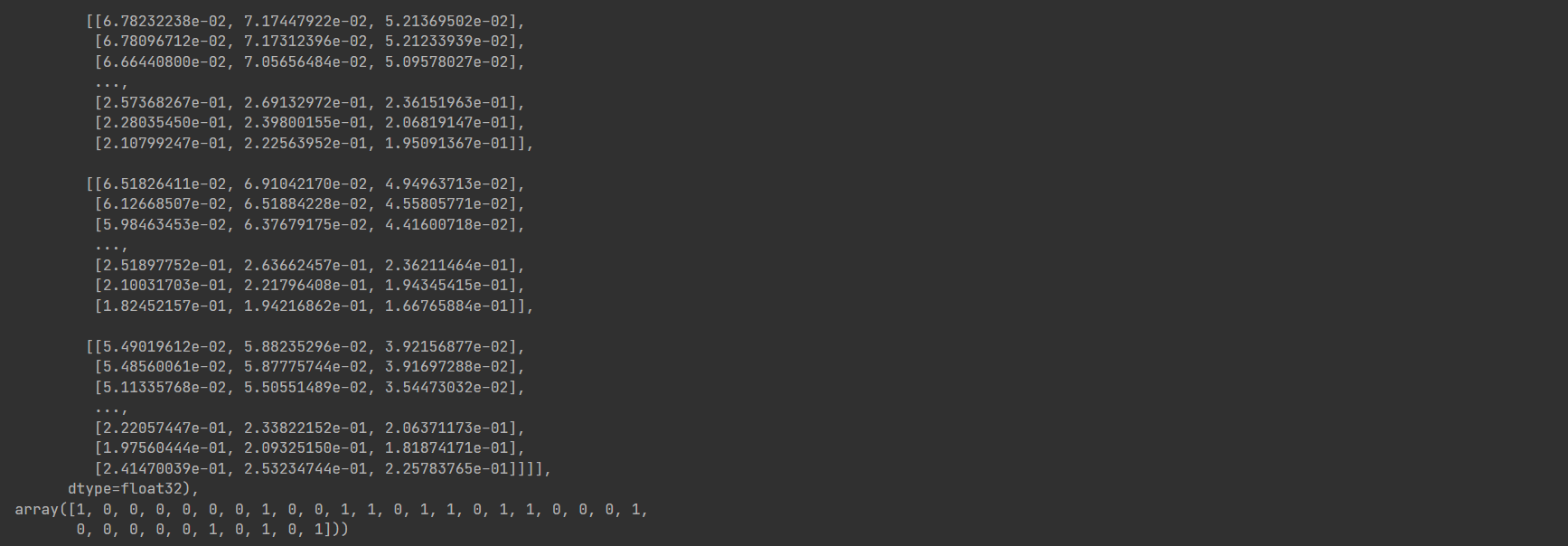
Dataset Generation: It creates a TensorFlow dataset directly from a directory structure containing images organized into subfolders representing their classes.

* NumPy Conversion: It transforms the TensorFlow dataset into a NumPy iterator for easier manipulation and interaction with other Python libraries.
* Batch Retrieval: It fetches a batch of images and their corresponding labels from the iterator, preparing them for visualization.
* Visualization Setup: It employs Matplotlib to create a multi-image figure with subplots, enabling visual inspection of the dataset.
* Image Display: It iterates through the first four images in the batch, displaying each image within its subplot and setting the subplot title to its corresponding label

# Scale Data





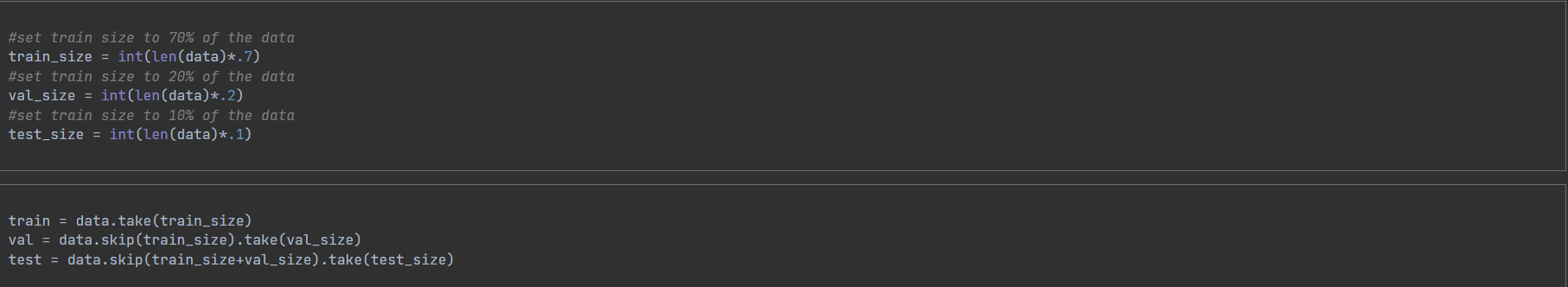


* Applying Normalization: It utilizes TensorFlow's map function to apply a normalization operation to each image and label pair within the dataset.
* Rescaling Pixel Values: The lambda function within map divides each pixel value in the image by 255.0, effectively rescaling them from the original range of 0-255 to a new range of 0-1. This ensures consistent input data distribution for the model.
* Fetching Normalized Batch: It demonstrates the normalization's effect by fetching the next batch of data from the dataset's NumPy iterator, now containing normalized pixel values.

This normalization technique contributes significantly to:

* Accelerated Convergence: Rescaled pixel values often lead to faster model convergence during training, as smaller input values can facilitate more efficient gradient updates.
* Improved Stability: Normalized inputs can enhance model stability and robustness, as they reduce the impact of outliers or large variations in pixel intensities.
* Enhanced Generalization: Consistent input ranges can contribute to better model generalization, enabling it to perform well on diverse image datasets.

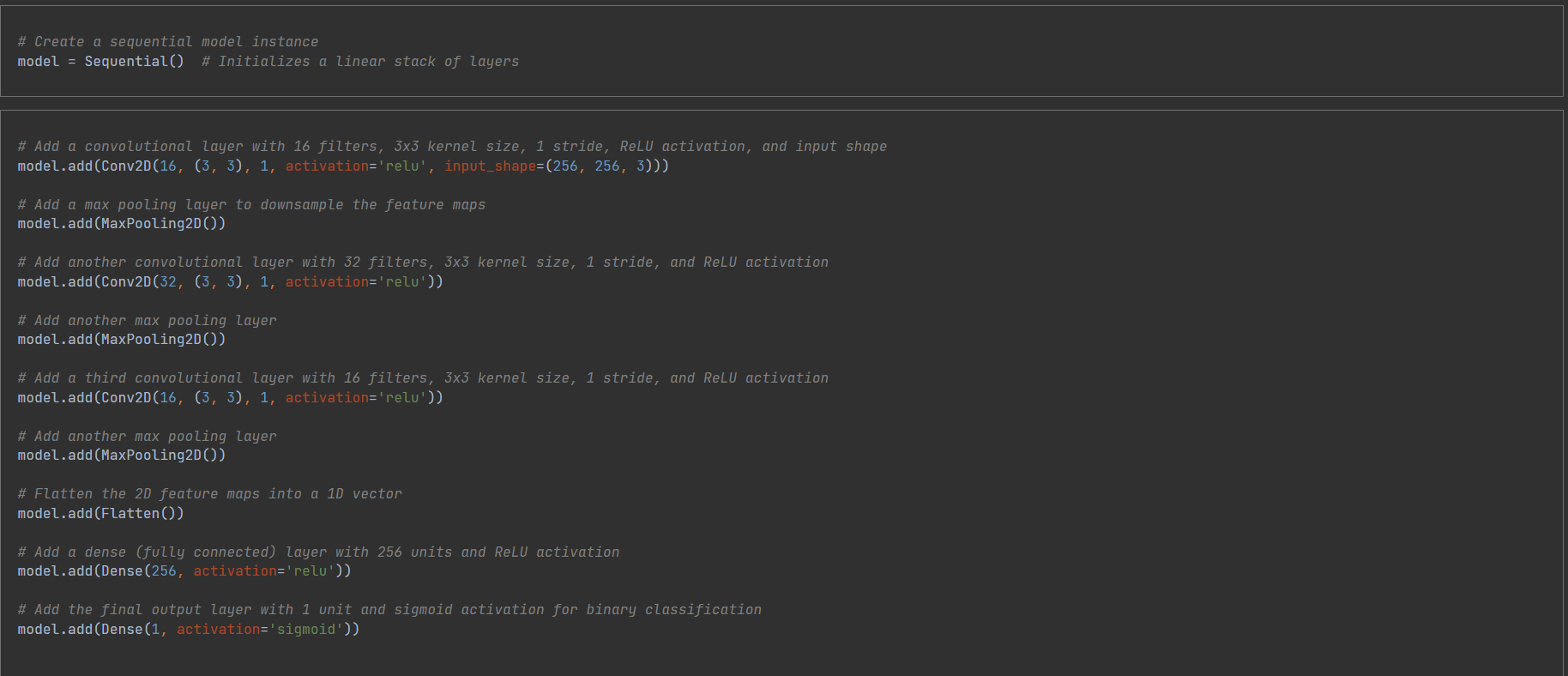
# Split Data



* Defining Subset Sizes: It establishes the desired proportions for each subset, allocating 70% of the data for training, 20% for validation, and 10% for testing.
* Creating Subsets: It leverages TensorFlow's dataset manipulation methods to create the subsets:
* Training Set: Composed of the first 70% of the dataset. Used to train the model, enabling it to learn patterns from the data.
* Validation Set: Captured by skipping the training set portion and taking the subsequent 20%. Employed to evaluate model performance during training and adjust hyperparameters, preventing overfitting.
* Testing Set: Formed by skipping both the training and validation sets and taking the remaining 10%. Reserved for final evaluation of the trained model's ability to generalize to unseen data.

This strategic splitting of the dataset ensures robust model development and unbiased performance assessment, cornerstones of successful image classification projects.

# Build Deep Learning Model (CNN)

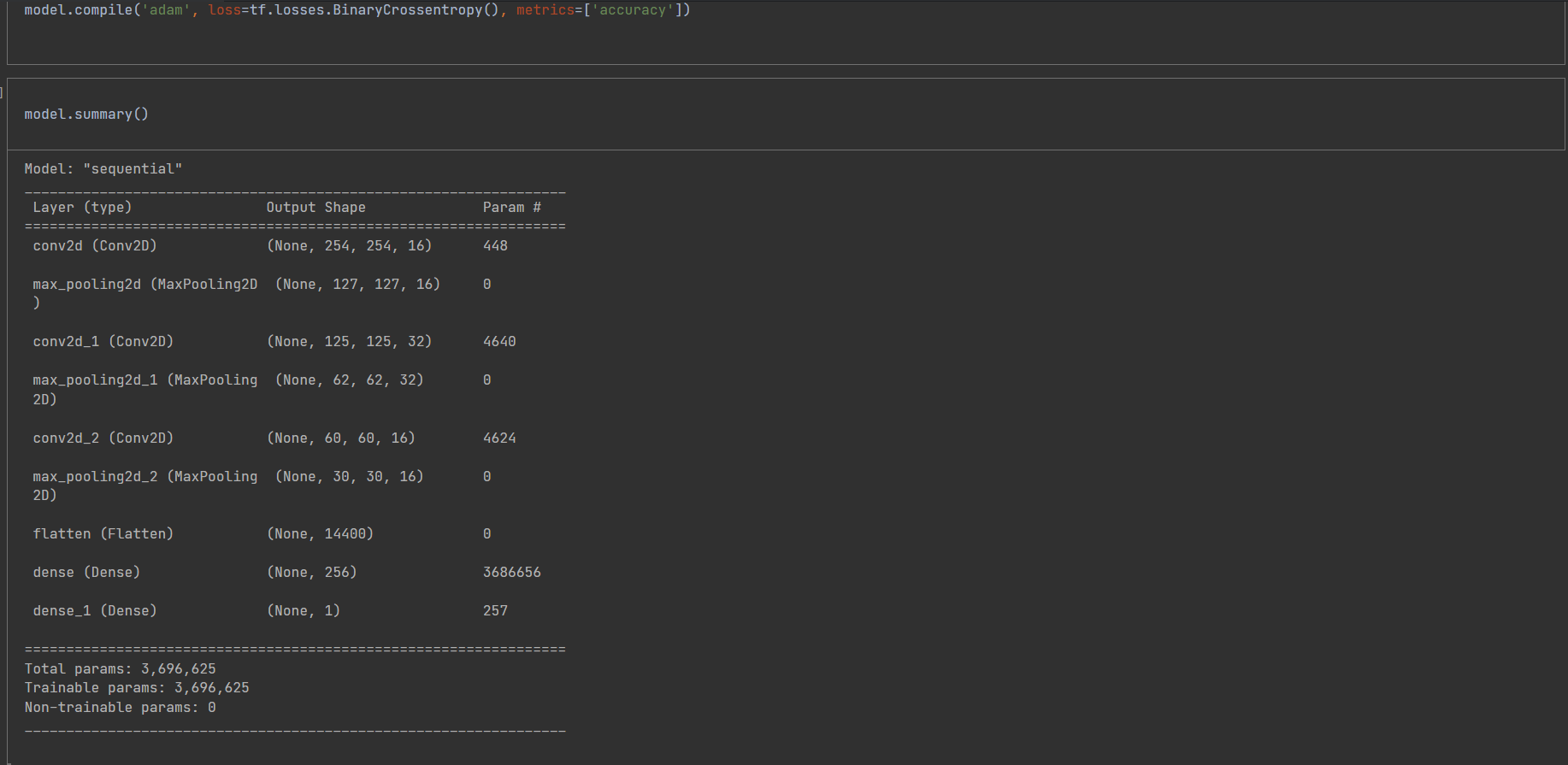


1. Sequential Model: It establishes a Sequential model, a linear stack of layers, forming the foundation for the CNN architecture.
2. Convolutional Layers: It incorporates a series of convolutional layers, each equipped with:

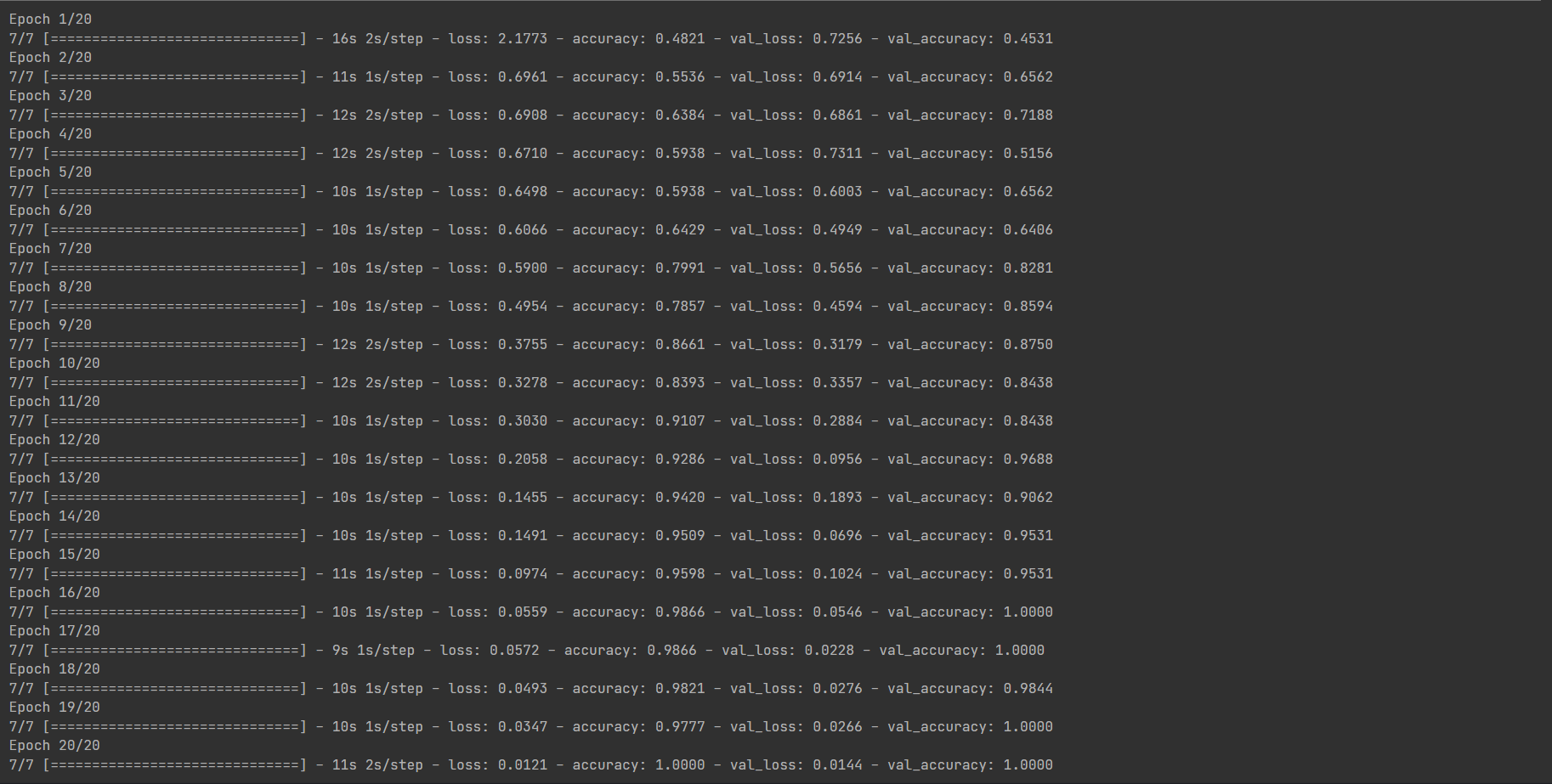
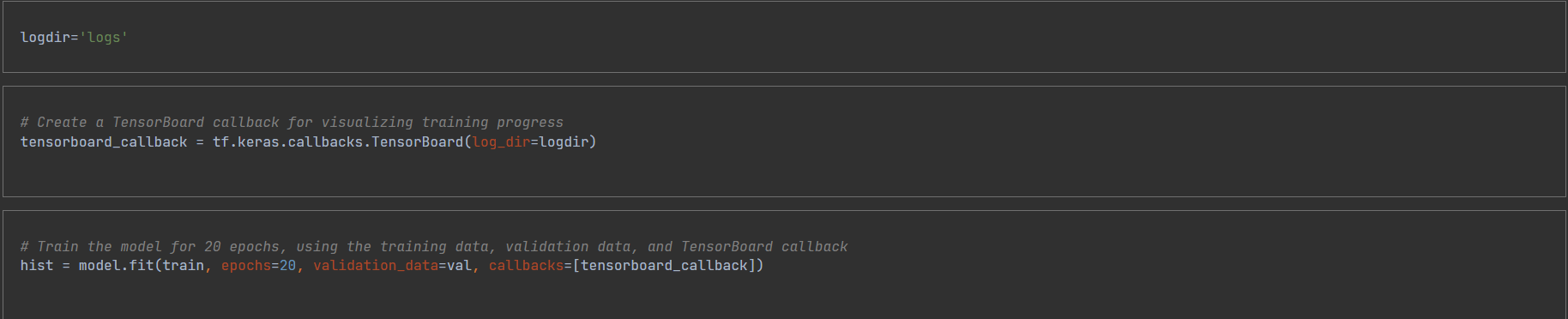
* Filters: These learnable kernels extract spatial features from images, capturing patterns like edges, shapes, and textures.
* Kernel Size: The 3x3 kernel size determines the receptive field of each filter, controlling the spatial context it considers.
* ReLU Activation: This non-linear activation function introduces non-linearity, allowing the model to learn complex patterns and preventing vanishing gradients.

1. Max Pooling: After each convolutional layer, a max pooling layer is added to downsample the feature maps, reducing computational complexity and introducing spatial invariance.
2. Flattening: The model then flattens the 2D feature maps into a 1D vector, preparing the data for further processing in fully connected layers.
3. Dense Layers: A dense layer with 256 units and ReLU activation follows, acting as a traditional neural network layer to further process the extracted features.
4. Output Layer: A final dense layer with a single unit and sigmoid activation is employed, producing a probability score between 0 and 1 for binary classification, indicating the likelihood of an image belonging to one of the two classes.

This carefully constructed CNN architecture effectively captures spatial features from images, enabling accurate classification of visual content.



# Train the Model

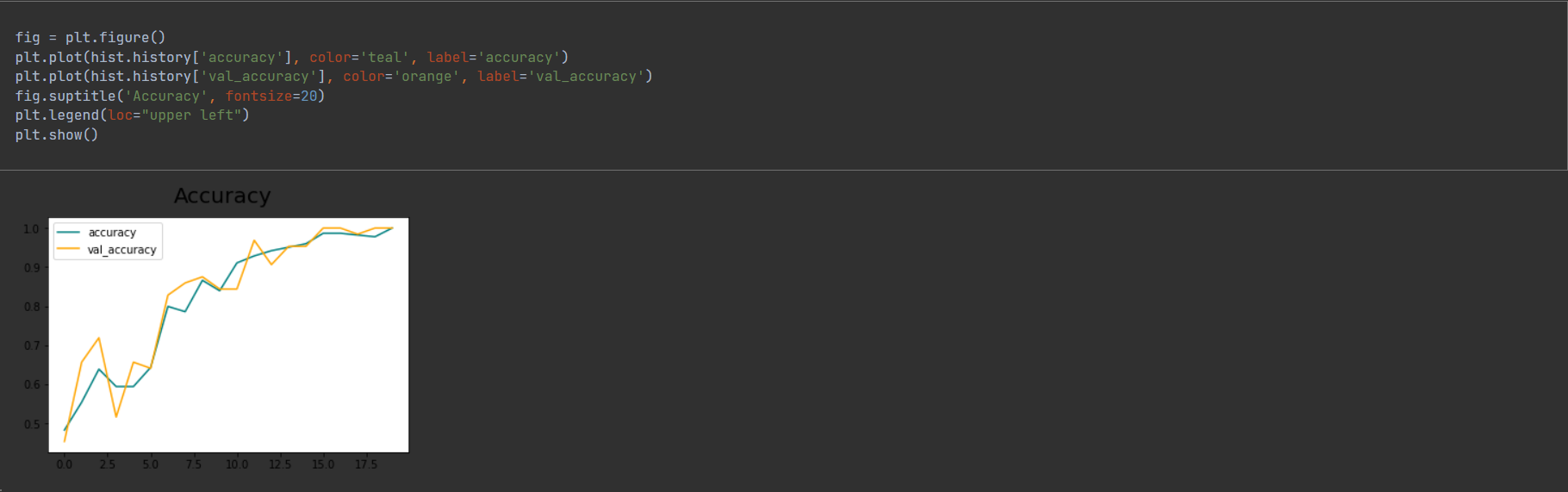


1. TensorBoard Setup: It creates a TensorBoard callback, a mechanism for visualizing training and evaluation metrics, aiding in understanding model behavior and identifying potential issues.
2. Log Directory: It specifies logdir='logs' as the directory to store TensorBoard logs, enabling creation of informative visualizations.
3. Model Training: It ignites the training process with model.fit(), guiding the model to learn from the provided image data:

* Training Data: It leverages the designated train dataset for model learning.
* Epochs: It sets the number of training epochs to 20, meaning the model will iterate through the entire training dataset 20 times.
* Validation Data: It incorporates the val dataset for evaluating model performance during training, helping to prevent overfitting and inform potential adjustments.
* TensorBoard Callback: It includes the TensorBoard callback, ensuring key metrics and visualizations are captured for analysis.

1. History Object: It stores the training history in the hist variable, preserving metrics for later exploration and evaluation.

# Plot Performance



1. Loss Visualization: It creates a plot showcasing how loss, a measure of the model's prediction errors, evolved during training:

* Training Loss: Depicted in teal, this curve reveals how well the model learned to fit the training data.
* Validation Loss: Illustrated in orange, this curve tracks model performance on unseen validation data, indicating its ability to generalize.

1. Accuracy Visualization: It constructs a plot highlighting accuracy, the proportion of correctly classified images, throughout training:

* Training Accuracy: Represented in teal, this curve reflects the model's ability to classify images within the training dataset.
* Validation Accuracy: Shown in orange, this curve demonstrates the model's performance on the validation set, assessing its ability to generalize to new data.

These visualizations offer valuable insights into the training process:

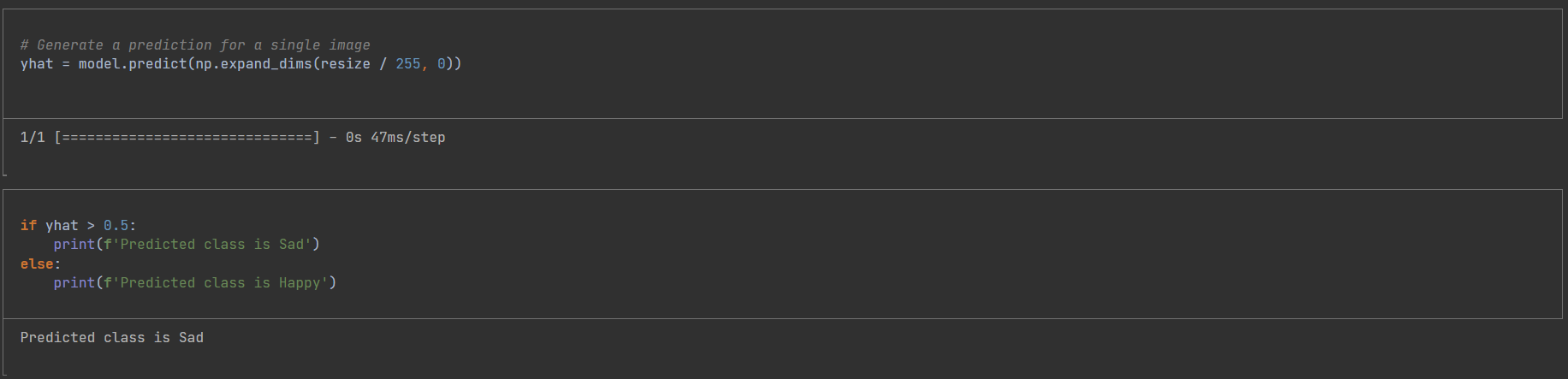
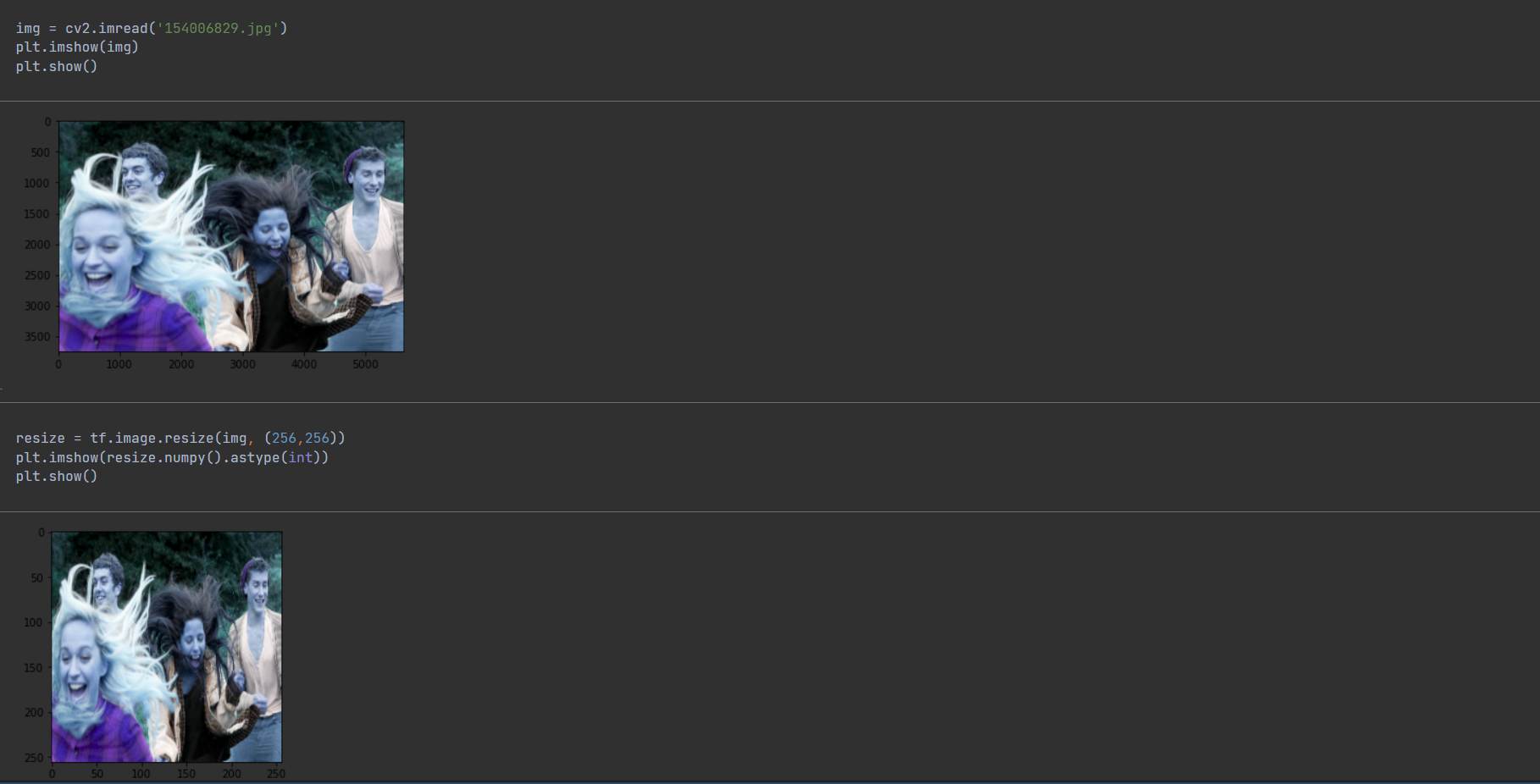
* General Trends: They reveal whether loss consistently decreased and accuracy increased, suggesting successful learning.
* Overfitting Detection: Divergence between training and validation curves can signal overfitting, where the model memorizes training data but struggles with new examples.
* Training Adjustments: These insights can guide decisions like adjusting hyperparameters or early stopping to optimize model performance.
* By visualizing these metrics, we gain a deeper understanding of the model's behavior and potential areas for improvement.

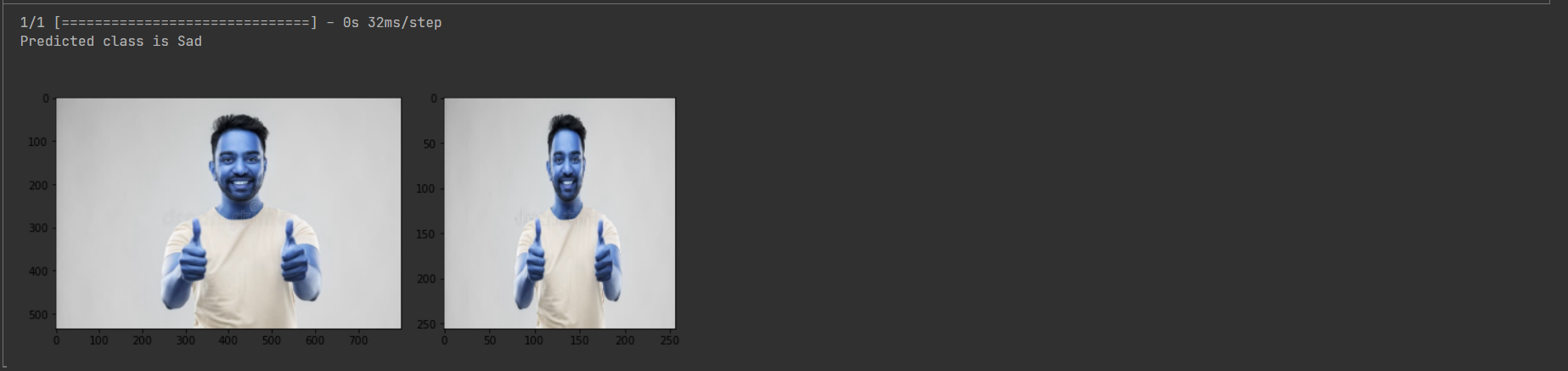
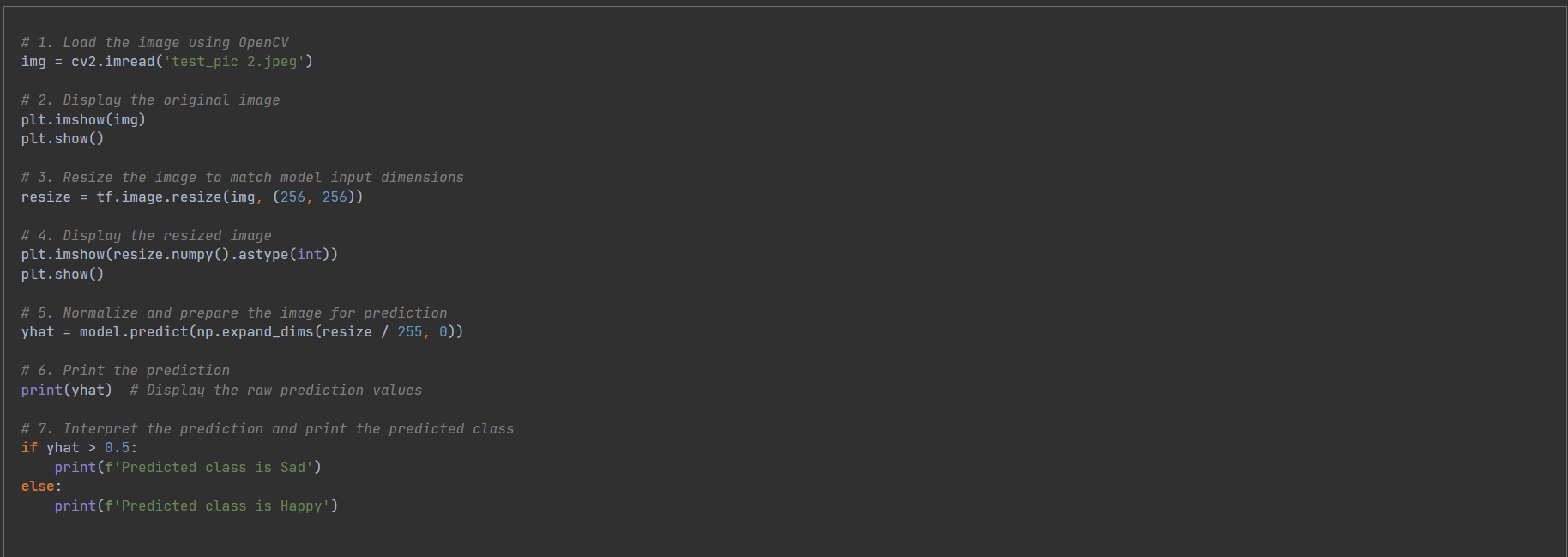
# Evaluate



* Metric Initialization: It prepares three essential metrics for evaluation:
* Precision: Measures the proportion of positive predictions that are truly positive, ensuring the model is accurate when it flags something as belonging to a certain class.
* Recall: Calculates the proportion of actual positive instances that are correctly identified as positive, ensuring the model doesn't miss true positives.
* BinaryAccuracy: Assesses the overall proportion of correctly classified images, providing a general performance overview.
* Test Evaluation: It systematically evaluates model performance on the test dataset, which hasn't been encountered during training:
* Batch Processing: It iterates through the test dataset in batches, enabling efficient computation.
* Prediction Generation: For each batch, it utilizes the model to generate predictions for the images, obtaining its classification decisions.
* Metric Updates: It updates the precision, recall, and accuracy metrics based on the true labels and model predictions, accumulating performance scores.
* Metric Results: It unveils the final evaluation results by printing the calculated precision, recall, and accuracy values, offering insights into the model's strengths and potential shortcomings.
* This comprehensive evaluation sheds light on the model's reliability and effectiveness in real-world image classification tasks, guiding decisions about its deployment and potential refinements.

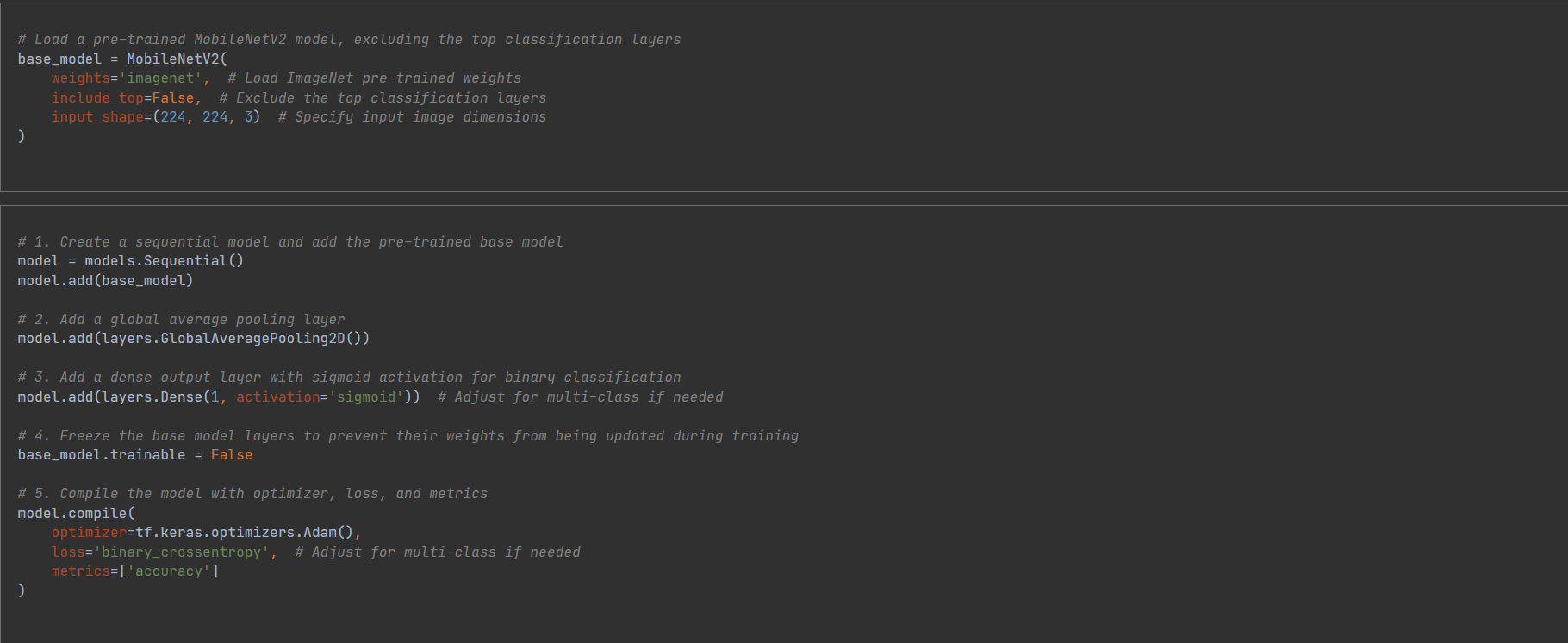
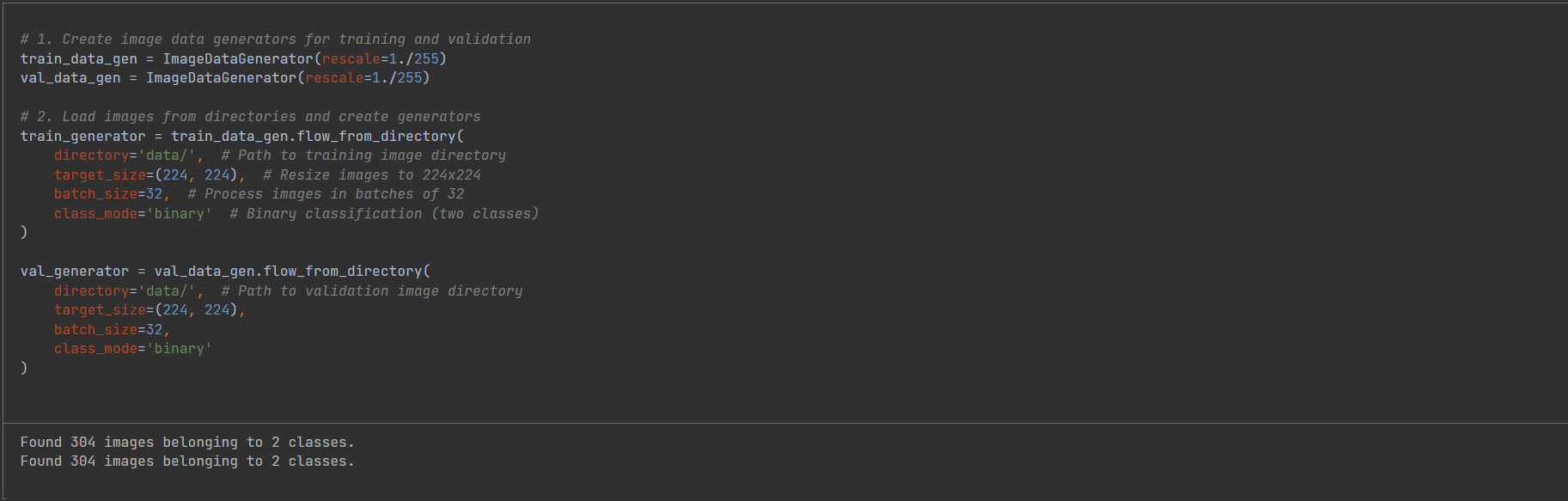
# Test

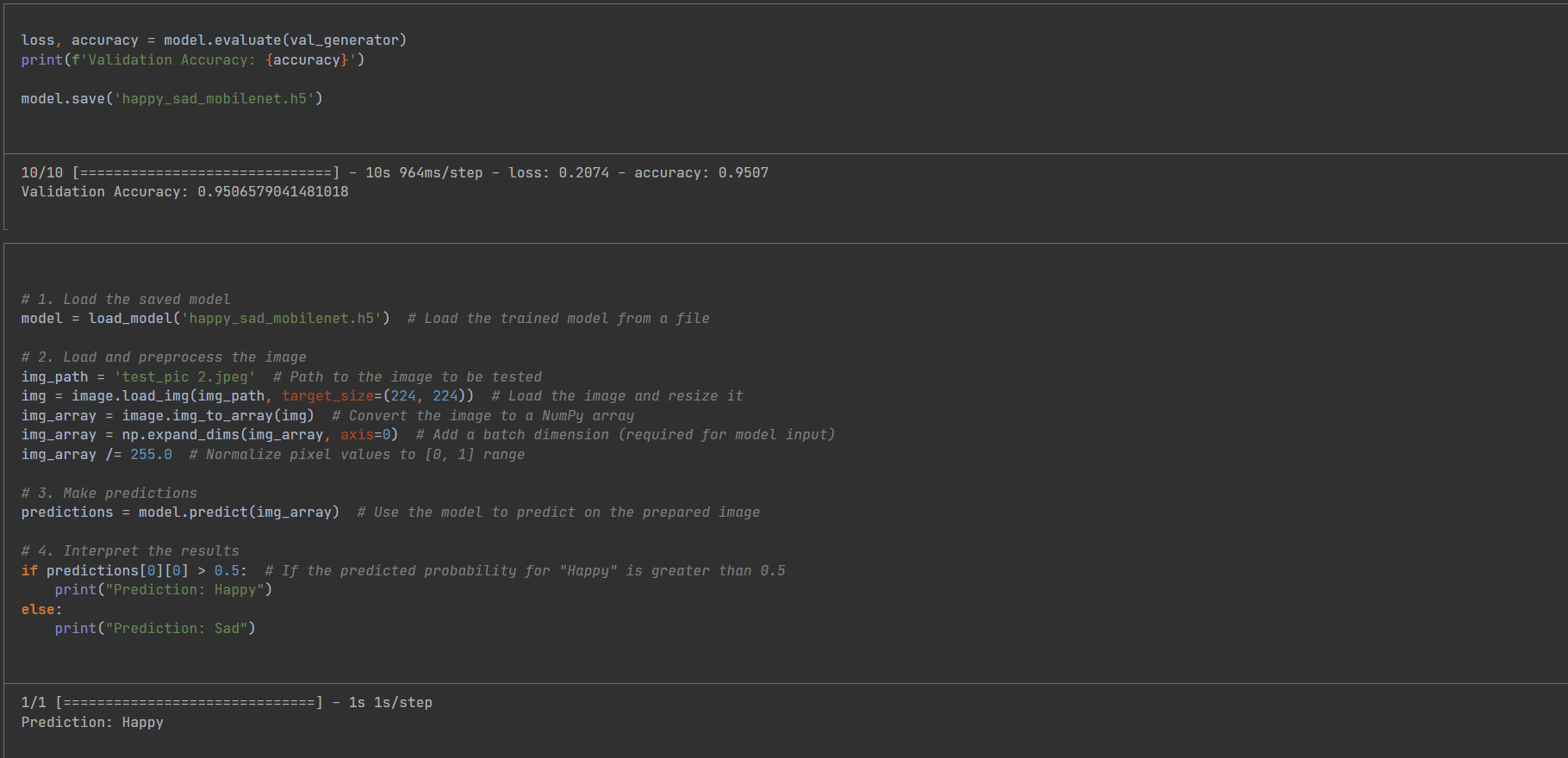
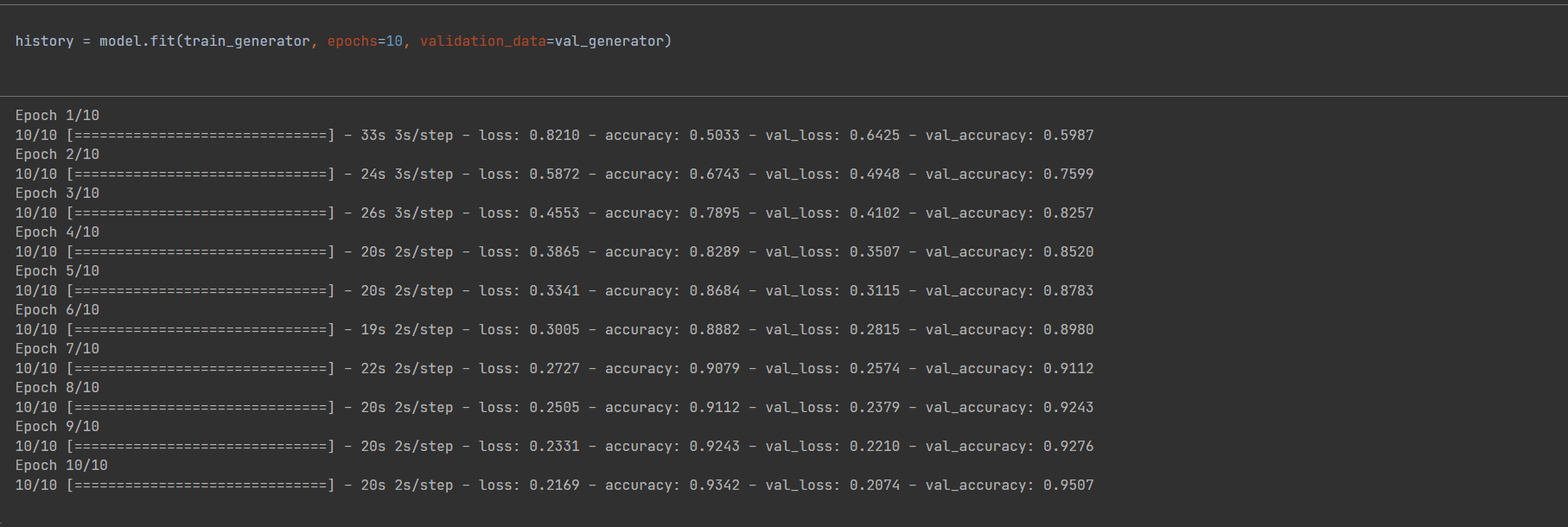
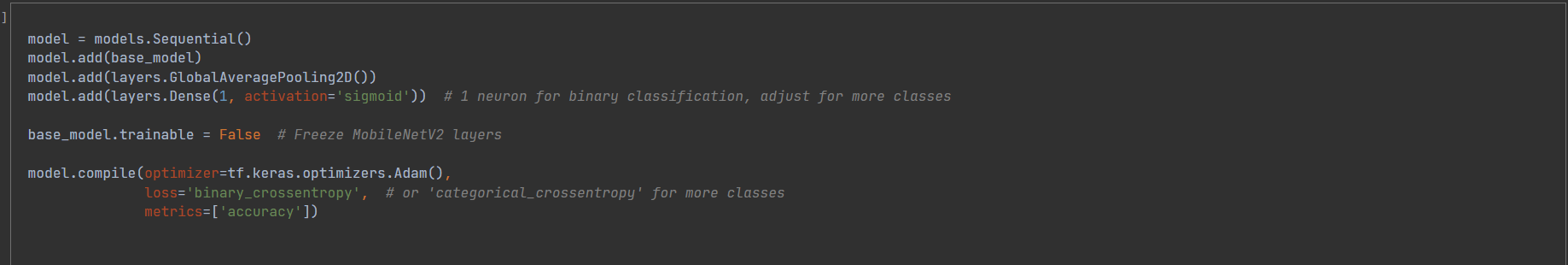




* Image Loading: It employs OpenCV's cv2.imread to load the image named 'test\_pic 2.jpeg', preparing it for further processing.
* Original Image Display: It utilizes Matplotlib's plt.imshow to visualize the original image, providing a visual reference for comparison with the resized version.
* Image Resizing: It resizes the image to 256x256 pixels using TensorFlow's tf.image.resize, ensuring it conforms to the input dimensions expected by the trained model.
* Resized Image Display: It visually presents the resized image using plt.imshow, demonstrating the impact of resizing and allowing for visual confirmation.
* Normalization and Prediction: It meticulously prepares the image for prediction:
* Normalization: It divides pixel values by 255.0 to normalize them to the range of 0-1, as expected by the model.
* Reshaping: It adds a dimension to the image using np.expand\_dims, transforming it into a batch of one image, a format compatible with the model's prediction function.
* Model Prediction: It employs model.predict to generate predictions based on the prepared image, obtaining the model's output.
* Raw Prediction Printing: It displays the raw prediction values, providing insights into the model's confidence in each class.
* Predicted Class Interpretation: It interprets the prediction based on a threshold of 0.5:
* Sad Class: If the prediction value exceeds 0.5, it indicates a higher probability of the image belonging to the "Sad" class.
* Happy Class: If the prediction value is less than or equal to 0.5, it suggests a higher probability of the image belonging to the "Happy" class.
* This process clearly outlines how to harness a trained model for classifying new images, providing a practical demonstration of its real-world application.

# Building deep learning model





# Alexnet

