Object Detection Model with Real-Time Capabilities Using Python

Lohit Marla

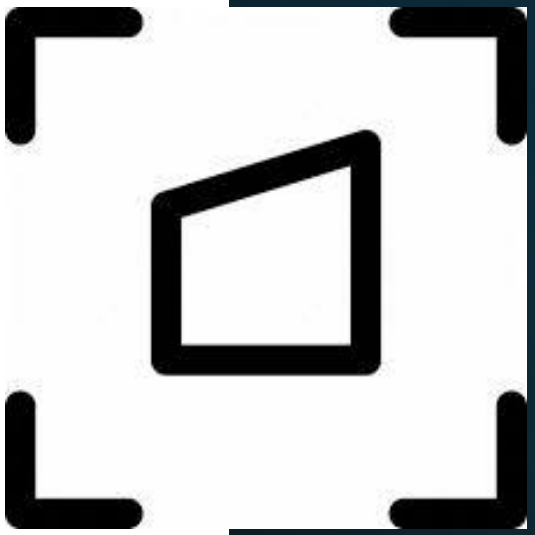
Agenda

1. Introduction to Object Detection
2. Problem Statement
3. End-to-End Model Development Pipeline
4. Workflow
5. Python Modules & Libraries
6. Model Selection and Python Code
    a) Pre-Trained Models
    b) Custom Models
7. Real-Time Video Frame Analysis Architecture
8. Detecting Objects & Bounding Boxes

Introduction to Object Detection

- Real-time object detection enables the identification and localization of objects within an image or video stream in real-time.

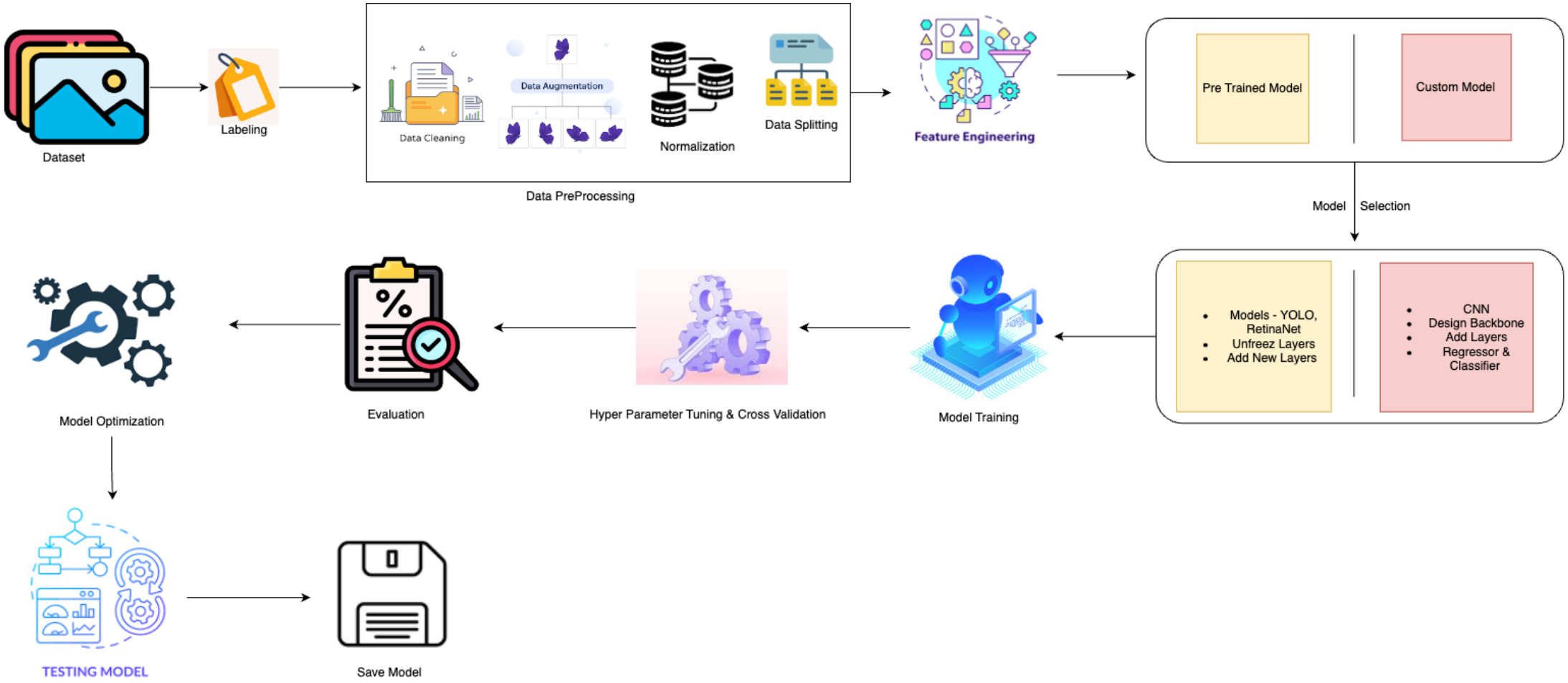
Example: Autonomous Vehicles, Surveillance Systems



Key Concepts

- Object Detection
- Bounding Boxes

End-to-End Model Development Pipeline



Workflow

Dataset Preparation

- The first step was to gather a dataset of images containing the objects of interest for the object detection model. For real-time object detection, we can leverage open datasets such as COCO & PASCAL VOC, which contain images annotated with bounding boxes and labels for various objects. Additionally, a small custom dataset can be created by manually labeling images relevant to the project.

Labeling

- Since object detection requires not only class labels but also the precise coordinates of the bounding boxes around the objects, the labeling process was critical. Tools such as LabelImg to annotate the custom dataset, drawing bounding boxes and assigning corresponding labels. Pre-labeled data from the COCO dataset was also utilized for standard objects.

Data Preprocessing

- The collected data was preprocessed by resizing the images to a uniform size, augmenting the dataset through transformations (such as flipping, rotation, and scaling) to increase the model's robustness. The bounding box coordinates were adjusted accordingly during augmentation. Additionally, the dataset should be normalized to ensure consistent input for the model.

Feature Engineering

- In this phase, enhancing the dataset to improve the model's performance is crucial. The object detection model required extracting specific features such as edges, corners, and textures from the images. Using techniques like Histogram of Oriented Gradients (HOG) and applying convolutional layers, model could recognize patterns relevant to object detection.

Model Selection

- For the real-time object detection task, several pre-trained models, including **MobileNetV2** and **YOLOv3**, that are optimized for both speed and accuracy. MobileNetV2 was selected for the initial experiment due to its lightweight architecture and efficiency in real-time scenarios. The model was fine-tuned to adapt to the custom dataset.

Model Training

- The selected model can be trained using the prepared dataset. During training, the transfer learning technique to leverage pre-trained weights from a general object detection task and applied them to a specific dataset. This significantly reduced training time while maintaining accuracy.

Hyperparameter Tuning

- To improve model performance, hyperparameters such as learning rate, batch size, number of epochs, and dropout rates can be used. Experiments to identify the best combination of these parameters, ensuring the model's optimal performance in real-time object detection scenarios.

Cross Validation

- To ensure that the model was not overfitting, k-fold cross-validation can be implemented. The dataset was split into training and validation sets, and the model's performance was evaluated on unseen data to ensure it generalizes well to new images.

Model Evaluation

- The model's performance can be evaluated using metrics such as **mean Average Precision (mAP)**, precision, recall, and **Intersection over Union (IoU)**. The bounding boxes predicted by the model were compared with the ground truth labels, and the IoU score was calculated to assess the model's accuracy.

Model Optimization

- Post-evaluation, the model can be optimized to enhance performance in real-time applications. Techniques such as model pruning and quantization can be applied to reduce the model size and inference time, allowing it to run efficiently on devices with limited computational resources.

Testing the Model

- The model can be tested in real-time using a webcam feed. Integrating the trained object detection model with OpenCV to capture live video streams and display bounding boxes and object labels over detected objects. The testing process involved verifying the model's ability to detect objects accurately and rapidly in real-world scenarios.

Saving the Model

- Once the model demonstrated satisfactory performance, it was saved for future use. The model's weights and architecture were stored using the TensorFlow/PyTorch save function, allowing it to be loaded and deployed for real-time object detection tasks at any time.

Python Modules & Libraries

1. Dataset Preparation	•os: For handling directory paths and file operations. •cv2 (OpenCV): For loading, processing, and visualizing images. •numpy: For handling arrays and numerical computations.
2. Labeling	•json: For handling JSON files (used for loading annotations in COCO or custom formats). •os: For file path management.
3. Data Preprocessing	•tensorflow.keras.preprocessing.image.ImageDataGenerator: For image augmentation and preprocessing. •cv2 (OpenCV): For resizing and augmenting images.
4. Feature Engineering	•tensorflow.keras.applications (VGG16, MobileNetV2, etc.): For loading pre-trained models and extracting features. •matplotlib.pyplot: For visualizing feature maps and other results. •tensorflow.keras.models: For building models.
5. Model Selection	•tensorflow: For deep learning model construction, loading pre-trained models, and transfer learning.

Python modules and libraries

6. Model Training

tensorflow: For model training and compiling.

matplotlib.pyplot: For plotting training and validation curves.

7. Hyperparameter Tuning and Cross-Validation

sklearn.model_selection.KFold: For k-fold cross-validation.

tensorflow: For managing model training during cross-validation.

8. Model Evaluation

tensorflow: For model evaluation.

sklearn.metrics (precision_score, recall_score): For calculating precision, recall, and other evaluation metrics.

9. Model Optimization

tensorflow.keras.optimizers (Adam): For using different optimizers and adjusting learning rates during fine-tuning.

10. Testing Model

tensorflow: For loading and using the trained model on unseen data.

cv2 (OpenCV): For visualizing images with bounding boxes and class labels.

11. Saving Model

tensorflow: For saving and loading the trained model in .h5 or .pb format.

Model Selection Strategy Pre-Trained Models

Small Dataset

- Utilize existing models trained on large datasets.
- Examples: YOLO, SSD, Faster R-CNN, MobileNetV2.
- Benefits:
  - Faster implementation.
  - High accuracy with minimal data.
- Use Cases:
  - Quick prototyping.
  - Applications with limited data availability.

Dataset Preparation

```
import os
import cv2
import numpy as np

# Define dataset paths
dataset_path = "path/to/dataset"
train_images_path = os.path.join(dataset_path, 'train')
test_images_path = os.path.join(dataset_path, 'test')

# Load the images
def load_images(image_dir):
    images = []
    for img_file in os.listdir(image_dir):
        img = cv2.imread(os.path.join(image_dir, img_file))
        if img is not None:
            images.append(img)
    return images

train_images = load_images(train_images_path)
test_images = load_images(test_images_path)

# Dataset ready for processing
print(f"Loaded {len(train_images)} training images and {len(test_images)} testing images.")
```

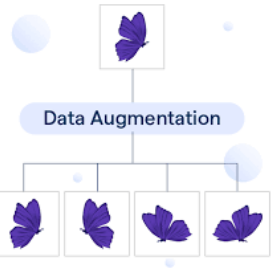
Labeling

```
# Assuming annotations are stored in JSON files in COCO format
import json

annotation_file = "path/to/annotations/train.json"

def load_annotations(annotation_path):
    with open(annotation_path, 'r') as file:
        annotations = json.load(file)
    return annotations

train_annotations = load_annotations(annotation_file)

# Example of annotation structure
for annotation in train_annotations['annotations']:
    print(annotation['bbox'], annotation['category_id'])
```

Data Preprocessing


Data Cleaning

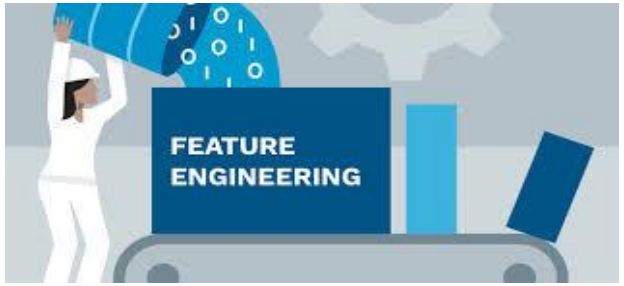
```
from tensorflow.keras.preprocessing.image import ImageDataGenerator

# Normalize and augment the data
def preprocess_data(images):
    # Create an image data generator for augmentations
    datagen = ImageDataGenerator(
        rescale=1./255,   # Normalize pixel values to [0, 1]
        rotation_range=10,
        zoom_range=0.2,
        horizontal_flip=True
    )

    processed_images = []
    for img in images:
        img = cv2.resize(img, (224, 224))  # Resize to match model input size
        img = datagen.random_transform(img)
        processed_images.append(img)

    return np.array(processed_images)

train_images_preprocessed = preprocess_data(train_images)
```




Normalization

Feature Engineering

```
# For object detection, feature extraction happens within the model's convolutional layers
# This step is mostly handled during model training
# However, we can visualize or extract feature maps from a pretrained model

from tensorflow.keras.applications import VGG16
import matplotlib.pyplot as plt

model = VGG16(weights='imagenet', include_top=False)
layer_outputs = [layer.output for layer in model.layers]

# Choose a specific image for feature map visualization
sample_img = train_images_preprocessed[0:1]  # Select one image from preprocessed

# Get feature maps for a specific layer
feature_model = tf.keras.Model(inputs=model.input, outputs=layer_outputs[5])  # 5th layer as an example
feature_maps = feature_model.predict(sample_img)

# Visualize the feature maps
plt.imshow(feature_maps[0, :, :, 0])
plt.show()
```



Model Selection

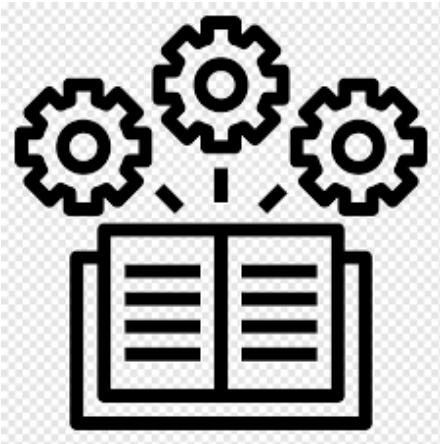
```
import tensorflow as tf
from tensorflow.keras.applications import MobileNetV2

# Load a pre-trained model for transfer learning
base_model = MobileNetV2(input_shape=(224, 224, 3), include_top=False, weights='imagenet')

# Add custom object detection head
model = tf.keras.Sequential([
    base_model,
    tf.keras.layers.GlobalAveragePooling2D(),
    tf.keras.layers.Dense(1024, activation='relu'),
    tf.keras.layers.Dense(80, activation='softmax')  # Assuming 80 object classes
])

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

Steps to Unfreeze Pre-Trained Model Layers & Add New Ones

```
# Unfreeze all layers
for layer in model.layers:
    layer.trainable = True

# Add new layers
x = base_model.output
x = tf.keras.layers.Dense(1024, activation='relu')(x)
x = tf.keras.layers.Dense(512, activation='relu')(x)
predictions = tf.keras.layers.Dense(num_classes, activation='softmax')(x)

# Create a new model
new_model = tf.keras.models.Model(inputs=base_model.input, outputs=predictions)

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

# Summary of the new model
new_model.summary()
```

Model Training

```
# Train the model on the dataset
history = model.fit(train_images_preprocessed, train_labels, epochs=10, batch_size=32, validation_split=0.2)

# Plot the training history (accuracy and loss)
import matplotlib.pyplot as plt

plt.plot(history.history['accuracy'], label='accuracy')
plt.plot(history.history['val_accuracy'], label = 'val_accuracy')
plt.xlabel('Epoch')
plt.ylabel('Accuracy')
plt.ylim([0, 1])
plt.legend(loc='lower right')
plt.show()
```



Hyperparameter Tuning and Cross-Validation

```
from sklearn.model_selection import KFold


kf = KFold(n_splits=5)

for train_index, val_index in kf.split(train_images_preprocessed):
    model.fit(train_images_preprocessed[train_index], train_labels[train_index],
              validation_data=(train_images_preprocessed[val_index], train_labels[val_index]),
              epochs=10)

    # Evaluate performance
    scores = model.evaluate(test_images, test_labels)
    print(f"Fold accuracy: {scores[1]}")
```

Model Evaluation

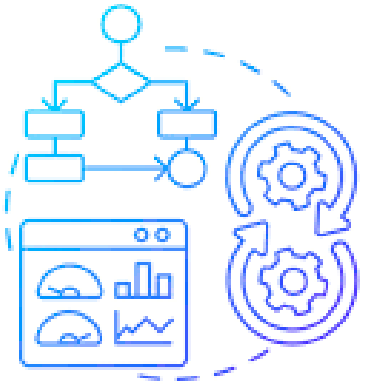
```
# Evaluate the model on the test set
test_loss, test_acc = model.evaluate(test_images, test_labels)

# Use Precision, Recall, and IoU for object detection evaluation
from sklearn.metrics import precision_score, recall_score

predictions = model.predict(test_images_preprocessed)
precision = precision_score(test_labels, predictions)
recall = recall_score(test_labels, predictions)

print(f"Test Accuracy: {test_acc}")
print(f"Precision: {precision}")
print(f"Recall: {recall}")
```

Model Optimization

```
# Example of fine-tuning the model
# Freeze all layers except the last few layers for fine-tuning
for layer in base_model.layers:
    layer.trainable = False

# Compile the model with a lower learning rate for fine-tuning
model.compile(optimizer=tf.keras.optimizers.Adam(learning_rate=1e-5),
              loss='categorical_crossentropy', metrics=['accuracy'])

model.fit(train_images_preprocessed, train_labels, epochs=5, batch_size=32)
```

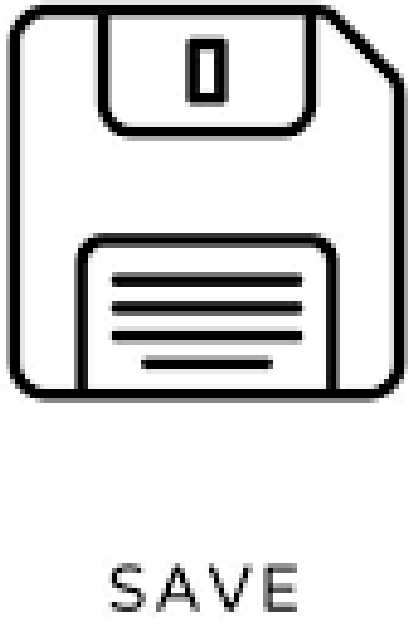
Testing Model

```
# Test the model on unseen data
unseen_test_images = load_images('path/to/unseen/test')
unseen_test_preprocessed = preprocess_data(unseen_test_images)

predictions = model.predict(unseen_test_preprocessed)

# Visualize predictions with bounding boxes and labels
for i, pred in enumerate(predictions):
    display_image_with_boxes(unseen_test_preprocessed[i], pred)
```



Save Model

```
# Save the model to a file
model.save('object_detection_model.h5')

# Load the model later
loaded_model =
tf.keras.models.load_model('object_detection
_model.h5')
```



Model Selection Strategy Custom Model

Large Dataset:

- Build and train models from scratch.
- Steps:
  - Data Collection and Annotation.
  - Model Architecture Design.
  - Training and Validation.
- Benefits:
  - Tailored to specific requirements.
  - Potential for higher accuracy with domain-specific data.
- Use Cases:
  - Specialized applications.
  - Scenarios requiring high customization

Dataset Preparation

```
import os
import cv2
import numpy as np

# Define dataset path and initialize lists
dataset_path = "/path/to/dataset"
image_files = []
labels = []

# Loop through dataset directory to load images and annotations
for filename in os.listdir(dataset_path):
    if filename.endswith(".jpg"):
        # Load the image
        image = cv2.imread(os.path.join(dataset_path, filename))
        image_files.append(image)

        # Load corresponding label (assume label is a .txt file with the same name)
        label_file = filename.replace(".jpg", ".txt")
        with open(os.path.join(dataset_path, label_file), 'r') as f:
            labels.append(f.read().strip())

print(f"Loaded {len(image_files)} images and labels.")
```

Labeling

```
import json

# Assuming labels are in COCO format (JSON)
with open("/path/to/annotations.json", "r") as f:
    annotations = json.load(f)

# Access image annotations
for annotation in annotations['annotations']:
    image_id = annotation['image_id']
    bbox = annotation['bbox']  # [x, y, width, height]
    category_id = annotation['category_id']

    # Print annotation details
    print(f"Image ID: {image_id}, BBox: {bbox}, Category: {category_id}")
```

Data Preprocessing

```
from tensorflow.keras.preprocessing.image import ImageDataGenerator

# Initialize image data generator for augmentation
datagen = ImageDataGenerator(
    rescale=1./255,
    rotation_range=20,
    width_shift_range=0.2,
    height_shift_range=0.2,
    zoom_range=0.2,
    horizontal_flip=True,
    fill_mode='nearest'
)

# Load a sample image and preprocess it
image = cv2.imread("/path/to/sample_image.jpg")
image = np.expand_dims(image, axis=0)  # Expand dimensions to match model input
augmented_image = datagen.flow(image)

print("Image preprocessed and augmented.")
```

Feature Engineering

```
from tensorflow.keras.applications import VGG16
from tensorflow.keras.models import Model

# Load VGG16 pre-trained model without the top classification layers
base_model = VGG16(weights='imagenet', include_top=False, input_shape=(224, 224, 3))

# Create a new model for feature extraction
model = Model(inputs=base_model.input, outputs=base_model.output)

# Extract features for a batch of images
features = model.predict(image_files)

print("Features extracted from pre-trained VGG16.")
```

Model Selection

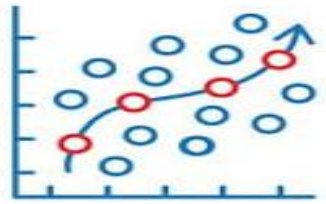
```
import tensorflow as tf
from tensorflow.keras import layers, models

# Define custom model architecture
model = models.Sequential([
    layers.Conv2D(32, (3, 3), activation='relu', input_shape=(224, 224, 3)),
    layers.MaxPooling2D((2, 2)),
    layers.Conv2D(64, (3, 3), activation='relu'),
    layers.MaxPooling2D((2, 2)),
    layers.Conv2D(128, (3, 3), activation='relu'),
    layers.Flatten(),
    layers.Dense(128, activation='relu'),
    layers.Dense(10, activation='softmax')  # Assuming 10 object classes
])

print("Custom model defined.")
```

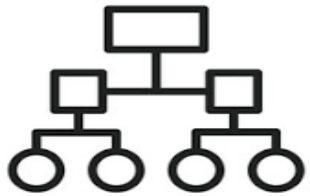
Backbone Feature Extractor

```
# Backbone feature extractor (could be ResNet, VGG, or custom CNN)
def backbone_model(input_shape):
    input_layer = layers.Input(shape=input_shape)
    x = layers.Conv2D(32, (3, 3), activation='relu')(input_layer)
    x = layers.MaxPooling2D((2, 2))(x)
    x = layers.Conv2D(64, (3, 3), activation='relu')(x)
    x = layers.MaxPooling2D((2, 2))(x)
    x = layers.Conv2D(128, (3, 3), activation='relu')(x)
    x = layers.MaxPooling2D((2, 2))(x)
    return Model(inputs=input_layer, outputs=x)
```

Regressor & Classifier

```
def bounding_box_regressor(feature_map):
    x = layers.Flatten()(feature_map)
    x = layers.Dense(256, activation='relu')(x)
    bbox_output = layers.Dense(4, name='bounding_box')(x)
                # Predict 4 coordinates (x_min, y_min, x_max, y_max)
    return bbox_output
```





```
def object_classifier(feature_map, num_classes):
    x = layers.Flatten()(feature_map)
    x = layers.Dense(256, activation='relu')(x)
    class_output = layers.Dense(num_classes, activation='softmax', name='class_label')(x)  # Softmax for class
prediction
    return class_output
```

Model Compilation

```
def object_detection_model(input_shape, num_classes):
    backbone = backbone_model(input_shape)

    # Pass the feature map to both regressor and classifier
    bbox_output = bounding_box_regressor(backbone.output)
    class_output = object_classifier(backbone.output, num_classes)

    return Model(inputs=backbone.input, outputs=[bbox_output, class_output])

# Example model creation
input_shape = (224, 224, 3)  # Input image shape
num_classes = 10  # Number of object classes
model = object_detection_model(input_shape, num_classes)

# Compile the model with loss for both bounding boxes (regressor) and class labels (classifier)
model.compile(optimizer='adam',
              loss={'bounding_box': 'mse', 'class_label': 'categorical_crossentropy'},
              metrics={'bounding_box': 'mse', 'class_label': 'accuracy'})
```

Model Training

```
# Assuming X_train and y_train are your training images and labels
history = model.fit(X_train, y_train, epochs=20, batch_size=32, validation_split=0.2)

print("Model trained.")
```

Hyperparameter Tuning and Cross-Validation

```
from sklearn.model_selection import KFold

# 5-fold cross-validation
kf = KFold(n_splits=5)

for train_index, val_index in kf.split(X_train):
    X_train_cv, X_val_cv = X_train[train_index], X_train[val_index]
    y_train_cv, y_val_cv = y_train[train_index], y_train[val_index]

    model.fit(X_train_cv, y_train_cv, validation_data=(X_val_cv, y_val_cv), epochs=20)

print("Hyperparameter tuning and cross-validation completed.")
```

Model Evaluation

```
# Evaluate the model on the test dataset
test_loss, test_acc = model.evaluate(X_test, y_test)
print(f"Test Accuracy: {test_acc * 100:.2f}%")
```

Model Optimization

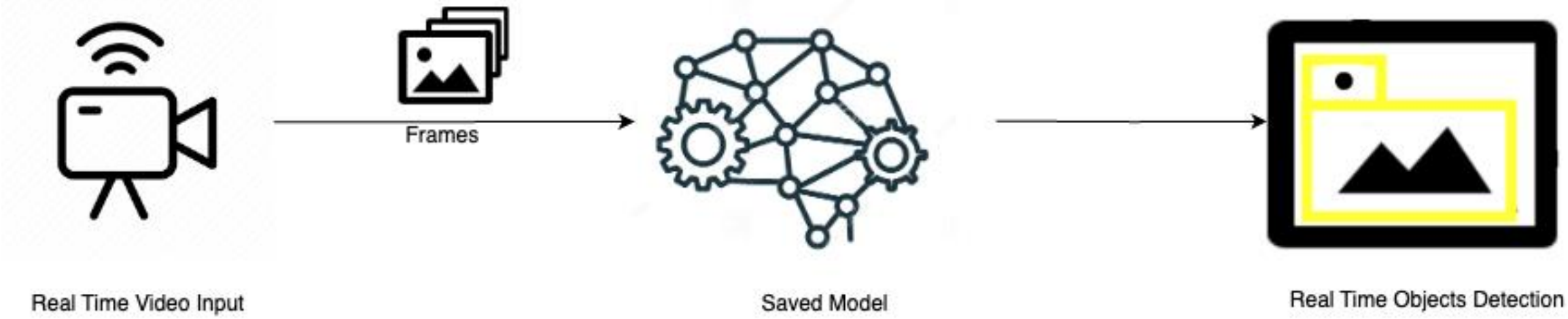
```
from tensorflow.keras.optimizers import Adam

# Use a custom learning rate for Adam optimizer
model.compile(optimizer=Adam(learning_rate=0.0001),
loss='categorical_crossentropy', metrics=['accuracy'])
model.fit(X_train, y_train, epochs=20, batch_size=32)

print("Model optimization complete.")
```

Testing Model

```
# Predict on new data
predictions = model.predict(X_test)

# Show predictions with bounding boxes (visualization with OpenCV)
for i, pred in enumerate(predictions):
    # Draw bounding boxes using OpenCV
    image = X_test[i]
    label = np.argmax(pred)
    cv2.rectangle(image, (x, y), (x+w, y+h), (0, 255, 0), 2)
    cv2.putText(image, f"Class: {label}", (x, y - 10), cv2.FONT_HERSHEY_SIMPLEX, 0.9, (36,255,12), 2)

cv2.imshow("Detected objects", image)
cv2.waitKey(0)
```

Save Model

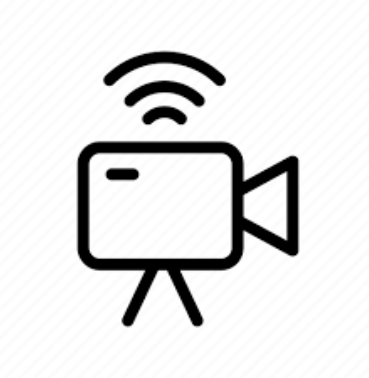
```
# Save model to disk
model.save("custom_object_detection_model.h5")
print("Model saved.")
```

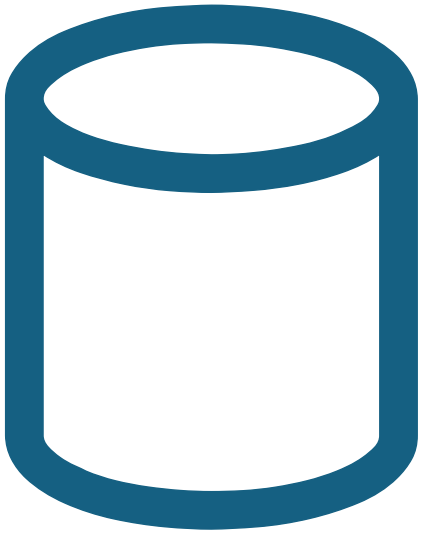


Real-Time Video Frame Analysis Architecture

Real-Time Video Capture Setup

```
import cv2

# Initialize the video capture object to capture real-time video from the webcam
cap = cv2.VideoCapture(0)  # '0' means the default webcam

# Check if the video capture is opened successfully
if not cap.isOpened():
    print("Error: Could not open video stream.")
    exit()

print("Video stream opened successfully.")
```





Loading the Pre-Trained Model

```
import tensorflow as tf

# Load the saved pre-trained object detection model
model = tf.keras.models.load_model('custom_object_detection_model.h5')

print("Pre-trained model loaded.")
```

Real-Time Frame Processing

```
import numpy as np

while True:
    # Capture each frame from the video
    ret, frame = cap.read()

    if not ret:
        print("Failed to capture video frame.")
        break

    # Preprocess the frame (resize and normalize)
    input_frame = cv2.resize(frame, (224, 224))  # Resize to the input size expected by the model
    input_frame = input_frame / 255.0  # Normalize the frame
    input_frame = np.expand_dims(input_frame, axis=0)  # Expand dimensions for batch processing

    print("Frame captured and preprocessed.")
```

Object Detection on Frames

```
# Perform object detection by predicting the frame
predictions = model.predict(input_frame)

# Extract the predicted class and bounding boxes (assuming single object detection)
predicted_class = np.argmax(predictions[0])
bbox = predictions[1]  # [x, y, w, h] format for bounding box

print(f"Predicted class: {predicted_class}, Bounding box: {bbox}")
```

Drawing Bounding Boxes

```
# Draw the bounding box and label on the original frame
x, y, w, h = bbox
cv2.rectangle(frame, (x, y), (x+w, y+h), (0, 255, 0), 2)  # Draw the rectangle
cv2.putText(frame, f"Class: {predicted_class}", (x, y - 10), cv2.FONT_HERSHEY_SIMPLEX, 0.9, (255, 0, 0), 2)  #
Add label

# Display the frame with bounding box
cv2.imshow('Real-Time Object Detection', frame)

print("Bounding box and label added to frame.")
```

Exiting the Video Stream

```
# Break the loop on 'q' key press
if cv2.waitKey(1) & 0xFF == ord('q'):
    break

# Release the video capture and close all OpenCV windows
cap.release()
cv2.destroyAllWindows()

print("Video stream ended.")
```

Thank You