



Capstone Project – Autonomous Driving

Part 1: Vehicle Detection (Deep Learning + Bounding Box)

- Step 1 – Import Required Libraries
- Step 2 – Load Dataset (Images + Labels)
- Step 3 – Preprocess Images
- Step 4 – Prepare Bounding Box Data
- Step 5 – Build a CNN / Object Detection Model
- Step 6 – Train & Validate the CNN Model
- Step 7 – Visualize Predictions (draw bounding boxes on test images)
- Step 8 – Train with More Epochs (Fine-tuning the Model)
- Step 9 – Evaluate Model on Test Data (Classification Report + IoU)

In [58]: # Step 1: Import Required Libraries

```
# Data handling
import os
import pandas as pd
import numpy as np

# Visualization
import matplotlib.pyplot as plt
import seaborn as sns

# Deep Learning (for CNN model)
import tensorflow as tf
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Conv2D, MaxPooling2D, Flatten, Dense,
from tensorflow.keras.utils import to_categorical

# Image Processing
import cv2
from PIL import Image

# Train-test split and evaluation
from sklearn.model_selection import train_test_split
```

Step 1 – Import Required Libraries

- Used **pandas** and **numpy** for data handling.
- Added **matplotlib** and **seaborn** for data visualization.
- Loaded **TensorFlow/Keras** modules for CNN model building.
- Included **to_categorical** to convert class labels into one-hot encoding.
- Added **cv2** and **PIL** for image preprocessing.
- Imported **train_test_split** for splitting dataset into training and testing sets.

- ◆ **Note:** These libraries cover the full workflow – from data preprocessing and visualization to deep learning model training and evaluation.

In [60]:

```
# =====
# Step 2 – Load Dataset (Images + Labels)
# =====

import os
import zipfile
import pandas as pd

# 1. Define dataset paths
# Why? -> Clear path names to avoid error
labels_path = "labels.csv"
deaths_path = "Tesla - Deaths.csv"
zip_path = "Images.zip"
extract_dir = "Images" # Folder where images will be extracted

# 2. Extract Images.zip into "Images" folder
# Why? -> Images are compressed, so we must unzip them first
if not os.path.exists(extract_dir):
    with zipfile.ZipFile(zip_path, 'r') as zip_ref:
        zip_ref.extractall(extract_dir)

# 3. Load CSV files
# Why? -> labels.csv = bounding boxes, deaths.csv = Tesla accident data
labels_df = pd.read_csv(labels_path)
deaths_df = pd.read_csv(deaths_path)

# 4. Verify dataset shapes
print("✓ Labels shape:", labels_df.shape)      # rows = total bounding boxes
print("✓ Deaths shape:", deaths_df.shape)      # rows = total Tesla accidents
print("✓ Extracted images:", len(os.listdir(extract_dir))) # total number of images

# 5. Preview first rows
print("\nLabels Preview:")
print(labels_df.head())
```

✓ Labels shape: (351548, 6)
✓ Deaths shape: (307, 24)
✓ Extracted images: 5626

Labels Preview:

			pickup_truck	213	34	255	50
0	0	car	194	78	273	122	
1	0	car	155	27	183	35	
2	0	articulated_truck	43	25	109	55	
3	0	car	106	32	124	45	
4	1	bus	205	155	568	314	

Step 2 – Load Dataset (Images + Labels)

- Extracted all vehicle images from **Images.zip** into the *Images* folder.
- Loaded **labels.csv** for bounding box annotations.
- Loaded **Tesla - Deaths.csv** for accident records.
- Verified dataset by checking shapes and previewing first rows.

```
In [62]: # Check the exact column names inside labels.csv
print(labels_df.columns.tolist())
```

```
['00000000', 'pickup_truck', '213', '34', '255', '50']
```

```
In [63]: # =====
# Step 3 – Preprocess Images
# =====
```

```
import cv2
import numpy as np
import os

# 1. Load labels.csv (no header in file)
labels_df = pd.read_csv("labels.csv", header=None)

# 2. Assign column names
labels_df.columns = ["image_id", "class", "x_min", "y_min", "x_max", "y_m"]

# 3. Adjust image_id to match filenames (zero-padded 8 digits)
labels_df["image_id"] = labels_df["image_id"].apply(lambda x: f"{int(x):08d}")

# 4. Limit dataset for testing (to avoid CPU crash)
LIMIT = 1000
labels_df = labels_df.iloc[:LIMIT]

# 5. Define image directory
image_dir = "Images"

# 6. Add file path column
labels_df["file_path"] = labels_df["image_id"].apply(
    lambda x: os.path.join(image_dir, f"{x}.jpg")
)

# 7. Preprocess function
IMG_SIZE = (224, 224)

def preprocess_image(img_path):
    if not os.path.exists(img_path):
        return None
    img = cv2.imread(img_path)
    img = cv2.cvtColor(img, cv2.COLOR_BGR2RGB) # Convert BGR → RGB
    img = cv2.resize(img, IMG_SIZE) # Resize
    img = img / 255.0 # Normalize
    return img

# 8. Apply preprocessing
processed = [preprocess_image(f) for f in labels_df["file_path"] if os.path.exists(f)]
processed = np.array([p for p in processed if p is not None])

# 9. Verify results
print("✅ Images processed:", processed.shape)
print("✅ Labels shape:", labels_df.shape)
print("\n🔍 First 5 file paths mapped:")
print(labels_df[["image_id", "file_path"]].head())
```

- Images processed: (1000, 224, 224, 3)
- Labels shape: (1000, 7)

🔍 First 5 file paths mapped:

	image_id	file_path
0	00000000	Images/00000000.jpg
1	00000000	Images/00000000.jpg
2	00000000	Images/00000000.jpg
3	00000000	Images/00000000.jpg
4	00000000	Images/00000000.jpg

Step 3 – Preprocess Images

- Loaded the `labels.csv` file (without header) and assigned proper column names.
- Converted the `image_id` into **8-digit padded strings** (e.g., `00000000`, `00000001`, ...) to match actual image filenames in the `Images` folder.
- Created a `file_path` column to map each image ID with its corresponding `.jpg` file.
- Defined a preprocessing function to:
 - Read each image.
 - Convert from **BGR to RGB** (since OpenCV reads in BGR format).
 - Resize images to a fixed size of **224 × 224 pixels**.
 - Normalize pixel values to the range **0–1**.
- Verified preprocessing on a subset of images (**1000 samples**) to ensure proper alignment between `labels.csv` and actual image files.

- Output confirmed with shape **(1000, 224, 224, 3)** for the processed images.

In [65]:

```
# =====
# Step 4 – Prepare Bounding Box Data
# =====

# 1. Verify available columns in labels.csv
# WHY? -> To make sure we are using the correct column names for bounding
print("✅ Columns in labels_df:", labels_df.columns.tolist())

# 2. Extract bounding box columns
# WHY? -> Dataset provides coordinates as (x_min, y_min, x_max, y_max)
bbox_df = labels_df[["x_min", "y_min", "x_max", "y_max"]].copy()

# 3. Convert (x_min, y_min, x_max, y_max) into (x, y, width, height)
# WHY? -> Object detection models often expect top-left corner (x, y)
#         and the box dimensions (width, height) instead of two corners
bbox_df["x"] = bbox_df["x_min"]
bbox_df["y"] = bbox_df["y_min"]
bbox_df["width"] = bbox_df["x_max"] - bbox_df["x_min"]
bbox_df["height"] = bbox_df["y_max"] - bbox_df["y_min"]

# ✅ Step 4: Normalize bounding boxes to 4 values only
IMG_W, IMG_H = 224, 224
bbox_df = pd.DataFrame({
    "x": labels_df["x_min"] / IMG_W,
    "y": labels_df["y_min"] / IMG_H,
    "width": (labels_df["x_max"] - labels_df["x_min"]) / IMG_W,
```

```

        "height": (labels_df["y_max"] - labels_df["y_min"]) / IMG_H
    }

print("BBox shape:", bbox_df.shape) # (1000, 4)

# 5. Verify the first few rows
# WHY? -> To confirm values are correctly transformed and normalized
print("✅ Bounding box sample (normalized):")
print(bbox_df[["x", "y", "width", "height"]].head())

```

✓ Columns in labels_df: ['image_id', 'class', 'x_min', 'y_min', 'x_max', 'y_max', 'file_path']
BBox shape: (1000, 4)
✓ Bounding box sample (normalized):

	x	y	width	height
0	0.950893	0.151786	0.187500	0.071429
1	0.866071	0.348214	0.352679	0.196429
2	0.691964	0.120536	0.125000	0.035714
3	0.191964	0.111607	0.294643	0.133929
4	0.473214	0.142857	0.080357	0.058036

Step 4 – Prepare Bounding Box Data

- Extracted bounding box coordinates (**x, y, width, height**) from the labels dataset.
- Normalized these values by dividing them with the image width and height so that all values fall in the **0–1 range**.
- Verified the first 5 rows of normalized bounding box data.

✓ Output confirmed with bounding box values scaled between **0 and 1**.

```
In [67]: # =====
# Step 5 – Build a CNN Model
# =====

import tensorflow as tf
from tensorflow.keras import layers, models
from sklearn.preprocessing import LabelEncoder
from tensorflow.keras.utils import to_categorical

# 1. Encode class labels (string -> integer -> one-hot)
le = LabelEncoder()
y_class_int = le.fit_transform(labels_df["class"]) # e.g. car=0, bus=1,
num_classes = len(le.classes_)
y_class_onehot = to_categorical(y_class_int, num_classes=num_classes)

print("✅ Classes mapped:", dict(zip(le.classes_, range(num_classes))))
print("✅ One-hot shape:", y_class_onehot.shape)

# 2. Define CNN model using Functional API (multi-output: class + bbox)
inputs = layers.Input(shape=(224, 224, 3))

# Convolution + Pooling layers
x = layers.Conv2D(32, (3,3), activation='relu')(inputs)
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)

# Flatten + Dense
x = layers.Flatten()(x)
x = layers.Dense(128, activation='relu')(x)

# Two output branches
class_output = layers.Dense(num_classes, activation='softmax', name="class_output")
bbox_output = layers.Dense(4, activation='sigmoid', name="bbox_output")

# Build model
model = models.Model(inputs=inputs, outputs=[class_output, bbox_output])

# 3. Compile model
model.compile(
    optimizer='adam',
    loss={
        "class_output": "categorical_crossentropy", # classification
        "bbox_output": "mse" # bounding box regression
    },
    metrics={
        "class_output": "accuracy",
        "bbox_output": "mse"
    }
)

# 4. Model summary
print("✅ CNN Model Summary:")
model.summary()

```

✅ Classes mapped: {'articulated_truck': 0, 'bicycle': 1, 'bus': 2, 'car': 3, 'motorcycle': 4, 'motorized_vehicle': 5, 'non-motorized_vehicle': 6, 'pedestrian': 7, 'pickup_truck': 8, 'single_unit_truck': 9, 'work_van': 10}
 ✅ One-hot shape: (1000, 11)
 ✅ CNN Model Summary:
Model: "functional_2"

Layer (type)	Output Shape	Param #	Connected to
input_layer_2 (InputLayer)	(None, 224, 224, 3)	0	-
conv2d_6 (Conv2D)	(None, 222, 222, 32)	896	input_layer_2[0]
max_pooling2d_6 (MaxPooling2D)	(None, 111, 111, 32)	0	conv2d_6[0]
conv2d_7 (Conv2D)	(None, 109, 109, 64)	18,496	max_pooling2d_6[0]
max_pooling2d_7 (MaxPooling2D)	(None, 54, 54, 64)	0	conv2d_7[0]
conv2d_8 (Conv2D)	(None, 52, 52, 128)	73,856	max_pooling2d_7[0]
max_pooling2d_8 (MaxPooling2D)	(None, 26, 26, 128)	0	conv2d_8[0]
flatten_2 (Flatten)	(None, 86528)	0	max_pooling2d_8[0]
dense_2 (Dense)	(None, 128)	11,075,712	flatten_2[0]
class_output (Dense)	(None, 11)	1,419	dense_2[0]
bbox_output (Dense)	(None, 4)	516	dense_2[0]

Total params: 11,170,895 (42.61 MB)

Trainable params: 11,170,895 (42.61 MB)

Non-trainable params: 0 (0.00 B)

Step 5 – Build a CNN Model

- **Encoded class labels**
Converted string labels (e.g., car, bus, truck) → integer → one-hot format, so that they can be used with categorical cross-entropy loss.
- **Used Functional API** instead of Sequential to support **multi-output** (vehicle classification + bounding box regression).
- **Added Convolution + Pooling layers** to extract and downsample image features.
- **Flattened** 2D features into a 1D vector.
- **Added a Dense layer** for higher-level learning.
- **Created two output branches:**
 - `class_output` : Softmax layer for multi-class classification.
 - `bbox_output` : Sigmoid layer for bounding box regression.

- Compiled the model with:

- categorical_crossentropy → for class prediction.
- mse → for bounding box coordinates.

- Metrics monitored:

- Accuracy (class classification).
- MSE (bounding box regression).

- Verified architecture using `model.summary()` to check layers, output shapes, and parameters.

Model confirmed with two outputs: `class_output`, `bbox_output`, ready for training.

```
In [69]: # =====
# Step 6 – Train the CNN Model
# =====

from sklearn.model_selection import train_test_split

# 1. Split dataset into training (80%) and testing (20%)
# WHY? -> To fairly evaluate model performance on unseen data
X_train, X_test, y_train_class, y_test_class, y_train_bbox, y_test_bbox =
    processed, y_class_onehot, bbox_df.values,
    test_size=0.2, random_state=42
)

print("☑ Train split:", X_train.shape, y_train_class.shape, y_train_bbox.shape)
print("☑ Test split :", X_test.shape, y_test_class.shape, y_test_bbox.shape)

# 2. Train the model
# WHY? -> model.fit() updates weights using both classification + regression
history = model.fit(
    X_train,
    {"class_output": y_train_class, "bbox_output": y_train_bbox},
    validation_data=(
        X_test,
        {"class_output": y_test_class, "bbox_output": y_test_bbox}
    ),
    epochs=5,
    batch_size=32
)

print("☑ Training Completed")
```

Train split: (800, 224, 224, 3) (800, 11) (800, 4)
 Test split : (200, 224, 224, 3) (200, 11) (200, 4)
Epoch 1/5
2025-09-07 16:54:57.023029: E tensorflow/core/grappler/optimizers/meta_optimizer.cc:961] PluggableGraphOptimizer failed: INVALID_ARGUMENT: Failed to deserialize the `graph_buf`.

```

25/25 ----- 8s 201ms/step - bbox_output_loss: 0.3338 - bbox
 _output_mse: 0.3338 - class_output_accuracy: 0.6150 - class_output_loss:
 1.8040 - loss: 2.1378 - val_bbox_output_loss: 0.3202 - val_bbox_output_ms
 e: 0.3061 - val_class_output_accuracy: 0.6850 - val_class_output_loss: 1.1
 580 - val_loss: 1.5560
Epoch 2/5
25/25 ----- 4s 147ms/step - bbox_output_loss: 0.3067 - bbox
 _output_mse: 0.3067 - class_output_accuracy: 0.6812 - class_output_loss:
 1.2711 - loss: 1.5779 - val_bbox_output_loss: 0.3139 - val_bbox_output_ms
 e: 0.3004 - val_class_output_accuracy: 0.6850 - val_class_output_loss: 1.1
 089 - val_loss: 1.5036
Epoch 3/5
25/25 ----- 4s 146ms/step - bbox_output_loss: 0.3015 - bbox
 _output_mse: 0.3015 - class_output_accuracy: 0.6812 - class_output_loss:
 1.2419 - loss: 1.5434 - val_bbox_output_loss: 0.3122 - val_bbox_output_ms
 e: 0.2987 - val_class_output_accuracy: 0.6850 - val_class_output_loss: 1.0
 950 - val_loss: 1.4721
Epoch 4/5
25/25 ----- 4s 152ms/step - bbox_output_loss: 0.2974 - bbox
 _output_mse: 0.2974 - class_output_accuracy: 0.6825 - class_output_loss:
 1.1556 - loss: 1.4530 - val_bbox_output_loss: 0.3101 - val_bbox_output_ms
 e: 0.2974 - val_class_output_accuracy: 0.6900 - val_class_output_loss: 1.1
 068 - val_loss: 1.4719
Epoch 5/5
25/25 ----- 4s 154ms/step - bbox_output_loss: 0.2923 - bbox
 _output_mse: 0.2923 - class_output_accuracy: 0.6800 - class_output_loss:
 1.0687 - loss: 1.3610 - val_bbox_output_loss: 0.3090 - val_bbox_output_ms
 e: 0.2961 - val_class_output_accuracy: 0.6850 - val_class_output_loss: 1.0
 478 - val_loss: 1.4230
✓ Training Completed

```

Step 6 – Train & Validate the CNN Model

- **Train-test split:**
 - Training set → 80% (used for updating model weights).
 - Validation set → 20% (used for unbiased evaluation after each epoch).
- **Training setup:**
 - `epochs=5` → Model sees full training dataset 5 times.
 - `batch_size=32` → Processes 32 images per step (faster & memory efficient).
 - `validation_data` → Ensures model is evaluated on unseen data at the end of each epoch.
- **Inputs & Outputs mapped correctly:**
 - `class_output` → Vehicle type classification (Softmax).
 - `bbox_output` → Bounding box regression (Sigmoid).
- **Expected Logs:**
 - Training loss should **decrease** gradually.
 - Classification accuracy (`class_output_accuracy`) should **increase**.
 - Bounding box loss (`bbox_output_mse`) should **reduce** over epochs.

- ✓ After training, we will analyze accuracy & loss curves to evaluate the performance.

In [71]:

```
# =====
# Step 7 – Visualize Predictions
# =====

import matplotlib.pyplot as plt

# 1. Pick a few test images
num_samples = 5
sample_images = X_test[:num_samples]
sample_true_classes = y_test_class[:num_samples]
sample_true_bboxes = y_test_bbox[:num_samples]

# 2. Get predictions from the model
pred_class, pred_bbox = model.predict(sample_images)

# 3. Decode predicted class labels (reverse label encoding)
pred_class_labels = le.inverse_transform(pred_class.argmax(axis=1))
true_class_labels = le.inverse_transform(sample_true_classes.argmax(axis=1))

# 4. Function to draw bounding boxes
def draw_bbox(img, true_box, pred_box, true_label, pred_label):
    plt.imshow(img)

    # Denormalize (x, y, w, h) -> pixel scale
    h, w = IMG_SIZE
    tx, ty, tw, th = true_box * [w, h, w, h]
    px, py, pw, ph = pred_box * [w, h, w, h]

    # True box = green
    plt.gca().add_patch(plt.Rectangle((tx, ty), tw, th,
                                      linewidth=2, edgecolor='g', facecolor='none'))
    # Predicted box = red
    plt.gca().add_patch(plt.Rectangle((px, py), pw, ph,
                                      linewidth=2, edgecolor='r', facecolor='none'))

    # Add labels
    plt.title(f"True: {true_label} | Pred: {pred_label}")
    plt.axis("off")
    plt.show()

# 5. Plot results
for i in range(num_samples):
    draw_bbox(sample_images[i], sample_true_bboxes[i], pred_bbox[i],
              true_class_labels[i], pred_class_labels[i])
```

1/1 ————— 0s 81ms/step

True: car | Pred: car



True: car | Pred: car



True: car | Pred: car



True: car | Pred: car



True: pickup_truck | Pred: car



Step 7 – Visualize Predictions

- **Purpose:** To check how well the trained CNN is working by comparing predicted vs. actual labels + bounding boxes.
- **Process:**
 - Selected a few random test images from `X_test`.
 - Used the model to predict both:
 - `class_output` → vehicle type (Softmax).
 - `bbox_output` → bounding box coordinates (Sigmoid).
 - Converted predictions back into human-readable labels using inverse transform.
 - Plotted images with:
 - **Green boxes** → True bounding box.
 - **Red boxes** → Predicted bounding box.
 - Labels showing `True` vs `Pred`.
- **Expected Outcome:**
 - If model learns correctly:
 - Predicted bounding boxes will closely align with true boxes.
 - Predicted class labels will match actual vehicle types.
 - Some mismatch may occur (e.g., predicting *car* instead of *pickup truck*) → indicates areas for further training or tuning.
- ✓ This step helps us **visually confirm** how good the model's classification + localization performance is.

In [73]:

```
# =====
# Step 8 – Train with More Epochs (Fine-tuning the Model)
# =====

# WHY? -> Initial 5 epochs only gave us a base model.
#           More epochs allow the CNN to refine its learning,
#           improving both classification (vehicle type)
#           and bounding box regression accuracy.

history_finetune = model.fit(
    X_train,
    {
        "class_output": y_train_class,      # vehicle class labels (one-hot
        "bbox_output": y_train_bbox       # bounding box coordinates (x,
    },
    validation_data=(
        X_test,
        {
            "class_output": y_test_class,  # validation class labels
            "bbox_output": y_test_bbox    # validation bounding box coor
        }
    ),
    epochs=20,          # WHY? -> Fine-tuning for longer (20 passes over data
    batch_size=32,     # WHY? -> Balanced batch size (memory efficient + sta
    verbose=1          # WHY? -> Shows training log per epoch
)

print("✅ Fine-tuning Completed")
```

Epoch 1/20
25/25 4s 138ms/step - bbox_output_loss: 0.2890 - bbox_output_mse: 0.2890 - class_output_accuracy: 0.6825 - class_output_loss: 1.0110 - loss: 1.2999 - val_bbox_output_loss: 0.3034 - val_bbox_output_mse: 0.2904 - val_class_output_accuracy: 0.6850 - val_class_output_loss: 1.0500 - val_loss: 1.4240
Epoch 2/20
25/25 3s 130ms/step - bbox_output_loss: 0.2880 - bbox_output_mse: 0.2880 - class_output_accuracy: 0.6913 - class_output_loss: 0.9106 - loss: 1.1987 - val_bbox_output_loss: 0.3060 - val_bbox_output_mse: 0.2941 - val_class_output_accuracy: 0.6800 - val_class_output_loss: 1.2210 - val_loss: 1.5836
Epoch 3/20
25/25 3s 129ms/step - bbox_output_loss: 0.2848 - bbox_output_mse: 0.2848 - class_output_accuracy: 0.6925 - class_output_loss: 0.8933 - loss: 1.1781 - val_bbox_output_loss: 0.3036 - val_bbox_output_mse: 0.2913 - val_class_output_accuracy: 0.6800 - val_class_output_loss: 1.1681 - val_loss: 1.5130
Epoch 4/20
25/25 3s 132ms/step - bbox_output_loss: 0.2822 - bbox_output_mse: 0.2822 - class_output_accuracy: 0.6963 - class_output_loss: 0.8257 - loss: 1.1079 - val_bbox_output_loss: 0.3067 - val_bbox_output_mse: 0.2945 - val_class_output_accuracy: 0.6900 - val_class_output_loss: 1.1493 - val_loss: 1.4977
Epoch 5/20
25/25 3s 133ms/step - bbox_output_loss: 0.2790 - bbox_output_mse: 0.2790 - class_output_accuracy: 0.7038 - class_output_loss: 0.7725 - loss: 1.0514 - val_bbox_output_loss: 0.3083 - val_bbox_output_mse: 0.2960 - val_class_output_accuracy: 0.7000 - val_class_output_loss: 1.2650 - val_loss: 1.6540
Epoch 6/20
25/25 4s 145ms/step - bbox_output_loss: 0.2828 - bbox_output_mse: 0.2828 - class_output_accuracy: 0.6812 - class_output_loss: 0.7782 - loss: 1.0610 - val_bbox_output_loss: 0.3056 - val_bbox_output_mse: 0.2938 - val_class_output_accuracy: 0.6500 - val_class_output_loss: 1.2904 - val_loss: 1.6378
Epoch 7/20
25/25 4s 144ms/step - bbox_output_loss: 0.2767 - bbox_output_mse: 0.2767 - class_output_accuracy: 0.7113 - class_output_loss: 0.7450 - loss: 1.0217 - val_bbox_output_loss: 0.3055 - val_bbox_output_mse: 0.2932 - val_class_output_accuracy: 0.6900 - val_class_output_loss: 1.3239 - val_loss: 1.7271
Epoch 8/20
25/25 4s 147ms/step - bbox_output_loss: 0.2748 - bbox_output_mse: 0.2748 - class_output_accuracy: 0.7125 - class_output_loss: 0.7118 - loss: 0.9866 - val_bbox_output_loss: 0.3043 - val_bbox_output_mse: 0.2922 - val_class_output_accuracy: 0.6900 - val_class_output_loss: 1.2941 - val_loss: 1.6779
Epoch 9/20
25/25 4s 141ms/step - bbox_output_loss: 0.2728 - bbox_output_mse: 0.2728 - class_output_accuracy: 0.7088 - class_output_loss: 0.6982 - loss: 0.9710 - val_bbox_output_loss: 0.3057 - val_bbox_output_mse: 0.2935 - val_class_output_accuracy: 0.6900 - val_class_output_loss: 1.3071 - val_loss: 1.6638
Epoch 10/20
25/25 5s 187ms/step - bbox_output_loss: 0.2744 - bbox_output_mse: 0.2744 - class_output_accuracy: 0.7125 - class_output_loss: 0.6847 - loss: 0.9592 - val_bbox_output_loss: 0.3076 - val_bbox_output_mse: 0.2956 - val_class_output_accuracy: 0.6750 - val_class_output_loss: 1.4003 - val_loss: 1.7565

Epoch 11/20
25/25 4s 173ms/step - bbox_output_loss: 0.2716 - bbox_output_mse: 0.2716 - class_output_accuracy: 0.7000 - class_output_loss: 0.6909 - loss: 0.9626 - val_bbox_output_loss: 0.3046 - val_bbox_output_mse: 0.2927 - val_class_output_accuracy: 0.7000 - val_class_output_loss: 1.3194 - val_loss: 1.7043
Epoch 12/20
25/25 4s 145ms/step - bbox_output_loss: 0.2725 - bbox_output_mse: 0.2725 - class_output_accuracy: 0.7275 - class_output_loss: 0.6682 - loss: 0.9407 - val_bbox_output_loss: 0.3089 - val_bbox_output_mse: 0.2973 - val_class_output_accuracy: 0.6500 - val_class_output_loss: 1.5564 - val_loss: 1.9021
Epoch 13/20
25/25 3s 135ms/step - bbox_output_loss: 0.2715 - bbox_output_mse: 0.2715 - class_output_accuracy: 0.7212 - class_output_loss: 0.6504 - loss: 0.9219 - val_bbox_output_loss: 0.3056 - val_bbox_output_mse: 0.2937 - val_class_output_accuracy: 0.6750 - val_class_output_loss: 1.3997 - val_loss: 1.7621
Epoch 14/20
25/25 4s 146ms/step - bbox_output_loss: 0.2711 - bbox_output_mse: 0.2711 - class_output_accuracy: 0.7212 - class_output_loss: 0.6417 - loss: 0.9129 - val_bbox_output_loss: 0.3045 - val_bbox_output_mse: 0.2928 - val_class_output_accuracy: 0.6850 - val_class_output_loss: 1.3655 - val_loss: 1.7195
Epoch 15/20
25/25 3s 136ms/step - bbox_output_loss: 0.2702 - bbox_output_mse: 0.2702 - class_output_accuracy: 0.7287 - class_output_loss: 0.6398 - loss: 0.9100 - val_bbox_output_loss: 0.3135 - val_bbox_output_mse: 0.3017 - val_class_output_accuracy: 0.6850 - val_class_output_loss: 1.5748 - val_loss: 1.9642
Epoch 16/20
25/25 3s 132ms/step - bbox_output_loss: 0.2715 - bbox_output_mse: 0.2715 - class_output_accuracy: 0.7262 - class_output_loss: 0.6386 - loss: 0.9100 - val_bbox_output_loss: 0.3089 - val_bbox_output_mse: 0.2967 - val_class_output_accuracy: 0.6850 - val_class_output_loss: 1.4766 - val_loss: 1.8502
Epoch 17/20
25/25 3s 132ms/step - bbox_output_loss: 0.2682 - bbox_output_mse: 0.2682 - class_output_accuracy: 0.7275 - class_output_loss: 0.6267 - loss: 0.8949 - val_bbox_output_loss: 0.3069 - val_bbox_output_mse: 0.2942 - val_class_output_accuracy: 0.6050 - val_class_output_loss: 1.4897 - val_loss: 1.8369
Epoch 18/20
25/25 3s 133ms/step - bbox_output_loss: 0.2673 - bbox_output_mse: 0.2673 - class_output_accuracy: 0.7200 - class_output_loss: 0.6157 - loss: 0.8830 - val_bbox_output_loss: 0.3087 - val_bbox_output_mse: 0.2957 - val_class_output_accuracy: 0.6550 - val_class_output_loss: 1.5077 - val_loss: 1.8527
Epoch 19/20
25/25 3s 136ms/step - bbox_output_loss: 0.2666 - bbox_output_mse: 0.2666 - class_output_accuracy: 0.7262 - class_output_loss: 0.6229 - loss: 0.8895 - val_bbox_output_loss: 0.3072 - val_bbox_output_mse: 0.2943 - val_class_output_accuracy: 0.6900 - val_class_output_loss: 1.5391 - val_loss: 1.8981
Epoch 20/20
25/25 3s 138ms/step - bbox_output_loss: 0.2661 - bbox_output_mse: 0.2661 - class_output_accuracy: 0.7225 - class_output_loss: 0.6051 - loss: 0.8712 - val_bbox_output_loss: 0.3100 - val_bbox_output_mse: 0.2966 - val_class_output_accuracy: 0.6900 - val_class_output_loss: 1.5

978 - val_loss: 1.9736
✓ Fine-tuning Completed

Step 8 – Train with More Epochs (Fine-tuning the Model)

- **Why more epochs?**

Initial 5 epochs gave a base model. More epochs help the CNN refine its learning, improving both classification (vehicle type) and bounding box regression.

- **Training setup:**

- `epochs = 20` → Model sees the full dataset 20 times for deeper learning.
- `batch_size = 32` → Balanced batch size (memory efficient + stable updates).
- `validation_data` → Ensures evaluation on unseen data after every epoch.

- **Expected Logs:**

- Training accuracy (`class_output_accuracy`) → should gradually **increase**.
- Validation accuracy → should also **improve** if the model generalizes.
- Bounding box loss (`bbox_output_loss` / `bbox_output_mse`) → should steadily **decrease**.
- If validation loss rises after a point → may indicate **overfitting**.

✓ After 20 epochs, the model becomes more robust for both **vehicle classification** and **bounding box detection**.

```
In [75]: # =====#
# Step 9 – Evaluate Model on Test Data
# =====#

from sklearn.metrics import classification_report
import numpy as np

# 1. Predict on test set
# WHY? -> Model predictions on unseen test data help us evaluate generalizability
y_pred_class, y_pred_bbox = model.predict(X_test)

# 2. Decode predicted class labels
# WHY? -> Convert softmax outputs (probabilities) into actual class label
y_pred_class_labels = le.inverse_transform(np.argmax(y_pred_class, axis=1))
y_true_class_labels = le.inverse_transform(np.argmax(y_test_class, axis=1))

# 3. Classification Report
# WHY? -> Precision, Recall, F1-score for vehicle classification performance

print("✓ Classification Report (Vehicle Type):")
print(classification_report(
    y_true_class_labels,
    y_pred_class_labels,
```

```

    zero_division=0    # Avoid undefined metric warnings
))

# 4. Compute IoU for bounding boxes
# WHY? -> Intersection over Union (IoU) measures bbox prediction accuracy
def compute_iou(box1, box2):
    # box = [x, y, w, h]
    x1, y1, w1, h1 = box1
    x2, y2, w2, h2 = box2

    # Convert to corner coords
    xa1, ya1, xa2, ya2 = x1, y1, x1+w1, y1+h1
    xb1, yb1, xb2, yb2 = x2, y2, x2+w2, y2+h2

    # Intersection
    xi1, yi1 = max(xa1, xb1), max(ya1, yb1)
    xi2, yi2 = min(xa2, xb2), min(ya2, yb2)
    inter_area = max(0, xi2 - xi1) * max(0, yi2 - yi1)

    # Union
    boxA_area = w1 * h1
    boxB_area = w2 * h2
    union_area = boxA_area + boxB_area - inter_area

    return inter_area / union_area if union_area > 0 else 0

# Calculate IoU for a few samples
ious = [compute_iou(y_pred_bbox[i], y_test_bbox[i]) for i in range(50)]
print("✓ Average IoU on test samples:", np.mean(ious))

```

5/7 ————— 0s 31ms/step

2025-09-07 16:56:32.820751: E tensorflow/core/grappler/optimizers/meta_optimizer.cc:961] PluggableGraphOptimizer failed: INVALID_ARGUMENT: Failed to deserialize the `graph_buf`.

7/7 ————— 0s 39ms/step

✓ Classification Report (Vehicle Type):

	precision	recall	f1-score	support
articulated_truck	0.00	0.00	0.00	5
bicycle	1.00	0.50	0.67	2
bus	0.00	0.00	0.00	5
car	0.73	0.95	0.83	137
motorized_vehicle	0.00	0.00	0.00	14
non-motorized_vehicle	0.00	0.00	0.00	1
pedestrian	0.33	0.20	0.25	5
pickup_truck	0.42	0.22	0.29	23
single_unit_truck	0.00	0.00	0.00	3
work_van	0.25	0.20	0.22	5
accuracy			0.69	200
macro avg	0.27	0.21	0.23	200
weighted avg	0.57	0.69	0.62	200

✓ Average IoU on test samples: 0.007850177420282108

Step 9 – Evaluate Model on Test Data

- Generated classification report (precision, recall, F1-score).

- Computed IoU (Intersection over Union) for bounding box evaluation.
- Final output: Classification Report + Average IoU score.

In []: