Main code:

```
import numpy as np
import tensorflow as tf
import cv2
import os
from PIL import Image
from distutils.version import StrictVersion
from collections import defaultdict
from object_detection.utils import ops as utils_ops
from utils import label_map_util
from utils import visualization_utils as vis_util
import sounddevice as sd
import librosa
import scipy.spatial
# Validate TensorFlow version
if StrictVersion(tf.__version__) < StrictVersion('1.9.0'):
    raise ImportError('Please upgrade your TensorFlow installation to v1.9.* or later!')
# --------- Image Detection Setup ---------
MODEL_NAME = 'inference_graph'
PATH_TO_FROZEN_GRAPH = MODEL_NAME + '/frozen_inference_graph.pb'
PATH_TO_LABELS = 'training/labelmap.pbtxt'
detection_graph = tf.Graph()
with detection_graph.as_default():
    od_graph_def = tf.GraphDef()
    with tf.gfile.GFile(PATH_TO_FROZEN_GRAPH, 'rb') as fid:
        serialized_graph = fid.read()
        od_graph_def.ParseFromString(serialized_graph)
        tf.import_graph_def(od_graph_def, name='')
```

```
category_index =
label_map_util.create_category_index_from_labelmap(PATH_TO_LABELS,
use_display_name=True)

def run_inference_for_single_image(image, sess):
    ops = tf.get_default_graph().get_operations()
    all_tensor_names = {output.name for op in ops for output in op.outputs}
    tensor_dict = {}
    for key in ['num_detections', 'detection_boxes', 'detection_scores', 'detection_classes',
'detection_masks']:
        tensor_name = key + ':0'
        if tensor_name in all_tensor_names:
            tensor_dict[key] = tf.get_default_graph().get_tensor_by_name(tensor_name)
    if 'detection_masks' in tensor_dict:
        detection_boxes = tf.squeeze(tensor_dict['detection_boxes'], [0])
        detection_masks = tf.squeeze(tensor_dict['detection_masks'], [0])
        real_num_detection = tf.cast(tensor_dict['num_detections'][0], tf.int32)
        detection_boxes = tf.slice(detection_boxes, [0, 0], [real_num_detection, -1])
        detection_masks = tf.slice(detection_masks, [0, 0, 0], [real_num_detection, -1, -1])
        detection_masks_reframed = utils_ops.reframe_box_masks_to_image_masks(
            detection_masks, detection_boxes, image.shape[0], image.shape[1])
        detection_masks_reframed = tf.cast(tf.greater(detection_masks_reframed, 0.5), tf.uint8)
        tensor_dict['detection_masks'] = tf.expand_dims(detection_masks_reframed, 0)
    image_tensor = tf.get_default_graph().get_tensor_by_name('image_tensor:0')
    output_dict = sess.run(tensor_dict, feed_dict={image_tensor: np.expand_dims(image, 0)})
    output_dict['num_detections'] = int(output_dict['num_detections'][0])
    output_dict['detection_classes'] = output_dict['detection_classes'][0].astype(np.uint8)
    output_dict['detection_boxes'] = output_dict['detection_boxes'][0]
    output_dict['detection_scores'] = output_dict['detection_scores'][0]
    if 'detection_masks' in output_dict:
        output_dict['detection_masks'] = output_dict['detection_masks'][0]
```

```
    if output_dict['detection_classes'][0] == 1 and output_dict['detection_scores'][0] > 0.70:
        print("🚑 Ambulance Detected in Image!")
    return output_dict
# --------- Audio Detection Setup ---------
def load_ambulance_sound(file_path="ambulance.mp3"):
    ref_audio, ref_sr = librosa.load(file_path, sr=22050)
    ref_mfcc = librosa.feature.mfcc(y=ref_audio, sr=ref_sr, n_mfcc=13)
    ref_mfcc_mean = np.mean(ref_mfcc, axis=1)
    return ref_mfcc_mean, ref_sr
def record_audio(duration=30, sr=22050):
    print("🎤 Listening for siren...")
    audio = sd.rec(int(duration * sr), samplerate=sr, channels=1, dtype=np.float32)
    sd.wait()
    print("✅ Audio recorded successfully")
    return np.squeeze(audio), sr
def compare_audio(ref_mfcc, ref_sr, test_audio, test_sr):
    test_mfcc = librosa.feature.mfcc(y=test_audio, sr=test_sr, n_mfcc=13)
    test_mfcc_mean = np.mean(test_mfcc, axis=1)
    ref_norm = ref_mfcc / np.linalg.norm(ref_mfcc)
    test_norm = test_mfcc_mean / np.linalg.norm(test_mfcc_mean)
    distance = scipy.spatial.distance.cosine(ref_norm, test_norm)
    print(f"🔍 Similarity Score (cosine distance): {distance:.3f}")
    return distance
def detect_siren():
    ref_mfcc, ref_sr = load_ambulance_sound()
    test_audio, test_sr = record_audio()
    similarity = compare_audio(ref_mfcc, ref_sr, test_audio, test_sr)
```

```
    if similarity < 0.2:
        print("🚨 Ambulance Siren Detected!")
    else:
        print("❌ No siren detected.")
# --------- Main Menu ---------
def main_menu():
    while True:
        print("\nChoose an option:")
        print("1. 🚑 Start Ambulance Image Detection (Press 'q' to quit)")
        print("2. 🎵 Start Ambulance Siren Detection")
        print("q. ❌ Quit")
        choice = input("Enter your choice: ")
        if choice == '1':
            cap = cv2.VideoCapture(0)
            with detection_graph.as_default():
                with tf.Session() as sess:
                    while True:
                        ret, frame = cap.read()
                        if not ret:
                            break
                        output_dict = run_inference_for_single_image(frame, sess)
                        vis_util.visualize_boxes_and_labels_on_image_array(
                            frame,
                            output_dict['detection_boxes'],
                            output_dict['detection_classes'],
                            output_dict['detection_scores'],
                            category_index,
                            instance_masks=output_dict.get('detection_masks'),
                            use_normalized_coordinates=True,
```

```
                    line_thickness=8)
                cv2.imshow("Ambulance Detection", cv2.resize(frame, (800, 600)))
                if cv2.waitKey(1) & 0xFF == ord('q'):
                    break
            cap.release()
            cv2.destroyAllWindows()
    elif choice == '2':
        detect_siren()
    elif choice.lower() == 'q':
        print("Exiting program.")
        break
    else:
        print("Invalid choice. Try again.")

if __name__ == "__main__":
    main_menu()
```

```
import os
import glob
import pandas as pd
import xml.etree.ElementTree as ET
```

Pascal VOC XML annotation files into CSV format

```
def xml_to_csv(path):
    xml_list = []
    for xml_file in glob.glob(path + '/*.xml'):
        tree = ET.parse(xml_file)
        root = tree.getroot()
        for member in root.findall('object'):
            value = (root.find('filename').text,
```

```
                int(root.find('size')[0].text),
                int(root.find('size')[1].text),
                member[0].text,
                int(member[4][0].text),
                int(member[4][1].text),
                int(member[4][2].text),
                int(member[4][3].text)
                )
            xml_list.append(value)
    column_name = ['filename', 'width', 'height', 'class', 'xmin', 'ymin', 'xmax', 'ymax']
    xml_df = pd.DataFrame(xml_list, columns=column_name)
    return xml_df


def main():

    for folder in ['train','test']:

        image_path = os.path.join(os.getcwd(), ('images/' + folder))

        xml_df = xml_to_csv(image_path)

        xml_df.to_csv(('images/' + folder + '_labels.csv'), index=None)

        print('Successfully converted xml to csv.')
main()
```

Training launcher for object detection models using TensorFlow's older Object Detection API

#
==========

r"""Training executable for detection models.

This executable is used to train DetectionModels. There are two ways of
configuring the training job:

1) A single pipeline_pb2.TrainEvalPipelineConfig configuration file
can be specified by --pipeline_config_path.

Example usage:

```
./train \
    --logtostderr \
    --train_dir=path/to/train_dir \
    --pipeline_config_path=pipeline_config.pbtxt
```

2) Three configuration files can be provided: a model_pb2.DetectionModel
configuration file to define what type of DetectionModel is being trained, an

```
input_reader_pb2.InputReader file to specify what training data will be used and
a train_pb2.TrainConfig file to configure training parameters.
Example usage:
    ./train \
        --logtostderr \
        --train_dir=path/to/train_dir \
        --model_config_path=model_config.pbtxt \
        --train_config_path=train_config.pbtxt \
        --input_config_path=train_input_config.pbtxt
"""
import functools
import json
import os
import tensorflow as tf
from object_detection.builders import dataset_builder
from object_detection.builders import graph_rewriter_builder
from object_detection.builders import model_builder
from object_detection.legacy import trainer
from object_detection.utils import config_util
tf.logging.set_verbosity(tf.logging.INFO)
flags = tf.app.flags
flags.DEFINE_string('master', '', 'Name of the TensorFlow master to use.')
flags.DEFINE_integer('task', 0, 'task id')
flags.DEFINE_integer('num_clones', 1, 'Number of clones to deploy per worker.')
flags.DEFINE_boolean('clone_on_cpu', False,
                     'Force clones to be deployed on CPU.  Note that even if '
                     'set to False (allowing ops to run on gpu), some ops may '
                     'still be run on the CPU if they have no GPU kernel.')
flags.DEFINE_integer('worker_replicas', 1, 'Number of worker+trainer '
```

```
                    'replicas.')
flags.DEFINE_integer('ps_tasks', 0,
                    'Number of parameter server tasks. If None, does not use '
                    'a parameter server.')
flags.DEFINE_string('train_dir', '',
                    'Directory to save the checkpoints and training summaries.')
flags.DEFINE_string('pipeline_config_path', '',
                    'Path to a pipeline_pb2.TrainEvalPipelineConfig config '
                    'file. If provided, other configs are ignored')
flags.DEFINE_string('train_config_path', '',
                    'Path to a train_pb2.TrainConfig config file.')
flags.DEFINE_string('input_config_path', '',
                    'Path to an input_reader_pb2.InputReader config file.')
flags.DEFINE_string('model_config_path', '',
                    'Path to a model_pb2.DetectionModel config file.')
FLAGS = flags.FLAGS
@tf.contrib.framework.deprecated(None, 'Use object_detection/model_main.py.')
def main(_):
  assert FLAGS.train_dir, '`train_dir` is missing.'
  if FLAGS.task == 0: tf.gfile.MakeDirs(FLAGS.train_dir)
  if FLAGS.pipeline_config_path:
    configs = config_util.get_configs_from_pipeline_file(
        FLAGS.pipeline_config_path)
    if FLAGS.task == 0:
      tf.gfile.Copy(FLAGS.pipeline_config_path,
                    os.path.join(FLAGS.train_dir, 'pipeline.config'),
                    overwrite=True)
  else:
    configs = config_util.get_configs_from_multiple_files(
```

```
      model_config_path=FLAGS.model_config_path,
      train_config_path=FLAGS.train_config_path,
      train_input_config_path=FLAGS.input_config_path)
  if FLAGS.task == 0:
    for name, config in [('model.config', FLAGS.model_config_path),
                         ('train.config', FLAGS.train_config_path),
                         ('input.config', FLAGS.input_config_path)]:
      tf.gfile.Copy(config, os.path.join(FLAGS.train_dir, name),
                    overwrite=True)
model_config = configs['model']
train_config = configs['train_config']
input_config = configs['train_input_config']
model_fn = functools.partial(
    model_builder.build,
    model_config=model_config,
    is_training=True)
def get_next(config):
  return dataset_builder.make_initializable_iterator(
      dataset_builder.build(config)).get_next()
create_input_dict_fn = functools.partial(get_next, input_config)
env = json.loads(os.environ.get('TF_CONFIG', '{}'))
cluster_data = env.get('cluster', None)
cluster = tf.train.ClusterSpec(cluster_data) if cluster_data else None
task_data = env.get('task', None) or {'type': 'master', 'index': 0}
task_info = type('TaskSpec', (object,), task_data)
# Parameters for a single worker.
ps_tasks = 0
worker_replicas = 1
worker_job_name = 'lonely_worker'
```

```
task = 0
is_chief = True
master = ''
if cluster_data and 'worker' in cluster_data:
  # Number of total worker replicas include "worker"s and the "master".
  worker_replicas = len(cluster_data['worker']) + 1
if cluster_data and 'ps' in cluster_data:
  ps_tasks = len(cluster_data['ps'])
if worker_replicas > 1 and ps_tasks < 1:
  raise ValueError('At least 1 ps task is needed for distributed training.')
if worker_replicas >= 1 and ps_tasks > 0:
  # Set up distributed training.
  server = tf.train.Server(tf.train.ClusterSpec(cluster), protocol='grpc',
                           job_name=task_info.type,
                           task_index=task_info.index)
  if task_info.type == 'ps':
    server.join()
    return
  worker_job_name = '%s/task:%d' % (task_info.type, task_info.index)
  task = task_info.index
  is_chief = (task_info.type == 'master')
  master = server.target
graph_rewriter_fn = None
if 'graph_rewriter_config' in configs:
  graph_rewriter_fn = graph_rewriter_builder.build(
      configs['graph_rewriter_config'], is_training=True)
trainer.train(
    create_input_dict_fn,
    model_fn,
```

```
      train_config,
      master,
      task,
      FLAGS.num_clones,
      worker_replicas,
      FLAGS.clone_on_cpu,
      ps_tasks,
      worker_job_name,
      is_chief,
      FLAGS.train_dir,
      graph_hook_fn=graph_rewriter_fn)
if __name__ == '__main__':
  tf.app.run()
```