Supplementary Materials

The core code in the manuscript

(1)Xml file

```
<annotation>
<folder>JPEGImages</folder>
<filename>(828).png</filename>
<path>C:\Users\lenovo\Desktop\VOCdevkit\VOC2007\JPEGImages\(828).png</path>
<source>
<database>Unknown</database>
</source>
<size>
<width>416</width>
<height>416</height>
<depth>3</depth>
</size>
<segmented>0</segmented>
<object>
<name>apple</name>
<pose>Unspecified</pose>
<truncated>0</truncated>
<difficult>0</difficult>
<bndbox>
<xmin>100</xmin>
<ymin>80</ymin>
<xmax>225</xmax>
<ymax>214</ymax>
</bndbox>
</object>
<object>
<name>apple</name>
<pose>Unspecified</pose>
<truncated>0</truncated>
<difficult>0</difficult>
<bndbox>
<xmin>81</xmin>
<ymin>214</ymin>
<xmax>205</xmax>
<ymax>347</ymax>
</bndbox>
</object>
<object>
<name>apple</name>
<pose>Unspecified</pose>
<truncated>0</truncated>
<difficult>0</difficult>
<bndbox>
<xmin>191</xmin>
<ymin>174</ymin>
<xmax>310</xmax>
<ymax>316</ymax>
</bndbox>
</object>
<object>
<name>apple</name>
<pose>Unspecified</pose>
<truncated>0</truncated>
<difficult>0</difficult>
<bndbox>
<xmin>219</xmin>
<ymin>72</ymin>
<xmax>325</xmax>
<ymax>189</ymax>
</bndbox>
</object>
<object>
```

```
<name>apple</name>
<pose>Unspecified</pose>
<truncated>1</truncated>
<difficult>0</difficult>
<bndbox>
<xmin>295</xmin>
<ymin>246</ymin>
<xmax>416</xmax>
<ymax>345</ymax>
</bndbox>
</object>
<object>
<name>apple</name>
<pose>Unspecified</pose>
<truncated>1</truncated>
<difficult>0</difficult>
<bndbox>
<xmin>328</xmin>
<ymin>127</ymin>
<xmax>416</xmax>
<ymax>251</ymax>
</bndbox>
</object>
</annotation>
```

(2) part code of the interface

```
import sys
import cv2
import time
import argparse
import random
import torch
import numpy as np
import torch.backends.cudnn as cudnn

from PyQt5 import QtCore, QtGui, QtWidgets
from PyQt5.QtCore import *
from PyQt5.QtGui import *
from PyQt5.QtWidgets import *

from utils.torch_utils import select_device
from models.experimental import attempt_load
from utils.general import check_img_size, non_max_suppression, scale_coords
from utils.datasets import letterbox
from utils.plots import plot_one_box2
from utils.torch_utils import select_device, load_classifier, time_synchronized

from ui.detect_ui import Ui_MainWindow

class UI_Logic_Window(QtWidgets.QMainWindow):
    def __init__(self, parent=None):
        super(UI_Logic_Window, self).__init__(parent)
        self.timer_video = QtCore.QTimer()
        self.ui = Ui_MainWindow()
        self.ui.setupUi(self)
        self.init_slots()
        self.cap = cv2.VideoCapture()
        self.num_stop = 1
        self.output_folder = 'output/'
        self.vid_writer = None


        self.openfile_name_model = None

    def init_slots(self):
        self.ui.pushButton_img.clicked.connect(self.button_image_open)
        self.ui.pushButton_video.clicked.connect(self.button_video_open)
        self.ui.pushButton_camer.clicked.connect(self.button_camera_open)
        self.ui.pushButton_weights.clicked.connect(self.open_model)
        self.ui.pushButton_init.clicked.connect(self.model_init)
        self.ui.pushButton_stop.clicked.connect(self.button_video_stop)
        self.ui.pushButton_finish.clicked.connect(self.finish_detect)

        self.timer_video.timeout.connect(self.show_video_frame)


    def open_model(self):
        self.openfile_name_model, _ = QFileDialog.getOpenFileName(self.ui.pushButton_weights, 'select weights file',
                                                                  'weights/')
        if not self.openfile_name_model:
            QtWidgets.QMessageBox.warning(self, u"Warning", u"fail", buttons=QtWidgets.QMessageBox.Ok,
                                          defaultButton=QtWidgets.QMessageBox.Ok)
        else:
            print(' weights file address：' + str(self.openfile_name_model))


    def model_init(self):
        parser = argparse.ArgumentParser()
        parser.add_argument('--weights', nargs='+', type=str, default='weights/yolov5s.pt', help='model.pt path(s)')
        parser.add_argument('--source', type=str, default='data/images', help='source')
        parser.add_argument('--img-size', type=int, default=640, help='inference size (pixels)')
        parser.add_argument('--conf-thres', type=float, default=0.25, help='object confidence threshold')
        parser.add_argument('--iou-thres', type=float, default=0.45, help='IOU threshold for NMS')
        parser.add_argument('--device', default='cpu', help='cuda device, i.e. 0 or 0,1,2,3 or cpu')
        parser.add_argument('--view-img', action='store_true', help='display results')
        parser.add_argument('--save-txt', action='store_true', help='save results to *.txt')
        parser.add_argument('--save-conf', action='store_true', help='save confidences in --save-txt labels')
        parser.add_argument('--nosave', action='store_true', help='do not save images/videos')
        parser.add_argument('--classes', nargs='+', type=int, help='filter by class: --class 0, or --class 0 2 3')
        parser.add_argument('--agnostic-nms', action='store_true', help='class-agnostic NMS')
        parser.add_argument('--augment', action='store_true', help='augmented inference')
        parser.add_argument('--update', action='store_true', help='update all models')
        parser.add_argument('--project', default='runs/detect', help='save results to project/name')
        parser.add_argument('--name', default='exp', help='save results to project/name')
        parser.add_argument('--exist-ok', action='store_true', help='existing project/name ok, do not increment')
        self.opt = parser.parse_args()
        print(self.opt)

        source, weights, view_img, save_txt, imgsz = self.opt.source, self.opt.weights, self.opt.view_img, self.opt.save_txt,
self.opt.img_size

        if self.openfile_name_model:
            weights = self.openfile_name_model
            print("Using button choose model")

        self.device = select_device(self.opt.device)
        self.half = self.device.type != 'cpu'

        cudnn.benchmark = True


        self.model = attempt_load(weights, map_location=self.device)
        stride = int(self.model.stride.max())
        self.imgsz = check_img_size(imgsz, s=stride)
        if self.half:
            self.model.half()
        classify = False
        if classify:
            modelc = load_classifier(name='resnet101', n=2)
            modelc.load_state_dict(torch.load('weights/resnet101.pt', map_location=self.device)['model']).to(self.device).eval()


        self.names = self.model.module.names if hasattr(self.model, 'module') else self.model.names
        self.colors = [[random.randint(0, 255) for _ in range(3)] for _ in self.names]
        print("model initial done")

        QtWidgets.QMessageBox.information(self, u"Notice", u"Completed", buttons=QtWidgets.QMessageBox.Ok,
                                  defaultButton=QtWidgets.QMessageBox.Ok)



def detect(self, name_list, img):

    showimg = img
    with torch.no_grad():
        img = letterbox(img, new_shape=self.opt.img_size)[0]

        img = img[:, :, ::-1].transpose(2, 0, 1)
        img = np.ascontiguousarray(img)
        img = torch.from_numpy(img).to(self.device)
        img = img.half() if self.half else img.float()
        img /= 255.0
        if img.ndimension() == 3:
            img = img.unsqueeze(0)

        pred = self.model(img, augment=self.opt.augment)[0]

        pred = non_max_suppression(pred, self.opt.conf_thres, self.opt.iou_thres, classes=self.opt.classes,
                                   agnostic=self.opt.agnostic_nms)
        info_show = ""

        for i, det in enumerate(pred):
            if det is not None and len(det):

                det[:, :4] = scale_coords(img.shape[2:], det[:, :4], showimg.shape).round()
                for *xyxy, conf, cls in reversed(det):
                    label = '%s %.2f' % (self.names[int(cls)], conf)
                    name_list.append(self.names[int(cls)])
                    single_info = plot_one_box2(xyxy, showimg, label=label, color=self.colors[int(cls)], line_thickness=2)

                    info_show = info_show + single_info + "\n"
    return  info_show

def button_image_open(self):
    print('button_image_open')
    name_list = []
    try:
        img_name, _ = QtWidgets.QFileDialog.getOpenFileName(self, "open image", "data/images", "*.jpg;;*.png;;All Files(*)")
    except OSError as reason:
        print('error！ image address'+ str(reason))
    else:

        if not img_name:
            QtWidgets.QMessageBox.warning(self, u"Warning", u"fail", buttons=QtWidgets.QMessageBox.Ok,
                                          defaultButton=QtWidgets.QMessageBox.Ok)
        else:
            img = cv2.imread(img_name)
            print("img_name:", img_name)
            info_show = self.detect(name_list, img)
            print(info_show)

            now = time.strftime("%Y-%m-%d-%H-%M-%S", time.localtime(time.time()))
            file_extension = img_name.split('.')[-1]
            new_filename = now + '.' + file_extension
            file_path = self.output_folder + 'img_output/' + new_filename
            cv2.imwrite(file_path, img)

            self.ui.textBrowser.setText(info_show)


            self.result = cv2.cvtColor(img, cv2.COLOR_BGR2BGRA)
            self.result = cv2.resize(self.result, (640, 480), interpolation=cv2.INTER_AREA)
            self.QtImg = QtGui.QImage(self.result.data, self.result.shape[1], self.result.shape[0], QtGui.QImage.Format_RGB32)
            self.ui.label.setPixmap(QtGui.QPixmap.fromImage(self.QtImg))
            self.ui.label.setScaledContents(True)
    def set_video_name_and_path(self):

        now = time.strftime("%Y-%m-%d-%H-%M-%S", time.localtime(time.time()))

        fps = self.cap.get(cv2.CAP_PROP_FPS)
        w = int(self.cap.get(cv2.CAP_PROP_FRAME_WIDTH))
        h = int(self.cap.get(cv2.CAP_PROP_FRAME_HEIGHT))

        save_path = self.output_folder + 'video_output/' + now + '.mp4'
        return fps, w, h, save_path

    def button_video_open(self):
        video_name, _ = QtWidgets.QFileDialog.getOpenFileName(self, "open video", "data/", "*.mp4;;*.avi;;All Files(*)")
        flag = self.cap.open(video_name)
        if not flag:
            QtWidgets.QMessageBox.warning(self, u"Warning", u"fail",
buttons=QtWidgets.QMessageBox.Ok,defaultButton=QtWidgets.QMessageBox.Ok)
        else:

            fps, w, h, save_path = self.set_video_name_and_path()
            self.vid_writer = cv2.VideoWriter(save_path, cv2.VideoWriter_fourcc(*'mp4v'), fps, (w, h))

            self.timer_video.start(30)

            self.ui.pushButton_video.setDisabled(True)
            self.ui.pushButton_img.setDisabled(True)
            self.ui.pushButton_camer.setDisabled(True)


    def button_camera_open(self):
        print("Open camera to detect")

        camera_num = 0

        self.cap = cv2.VideoCapture(camera_num)

        bool_open = self.cap.isOpened()
        if not bool_open:
            QtWidgets.QMessageBox.warning(self, u"Warning", u"fail", buttons=QtWidgets.QMessageBox.Ok,
                                          defaultButton=QtWidgets.QMessageBox.Ok)
        else:
            fps, w, h, save_path = self.set_video_name_and_path()
            fps = 5
            self.vid_writer = cv2.VideoWriter(save_path, cv2.VideoWriter_fourcc(*'mp4v'), fps, (w, h))
            self.timer_video.start(30)
            self.ui.pushButton_video.setDisabled(True)
            self.ui.pushButton_img.setDisabled(True)
            self.ui.pushButton_camer.setDisabled(True)


    def show_video_frame(self):
        name_list = []
        flag, img = self.cap.read()
        if img is not None:
            info_show = self.detect(name_list, img)
            self.vid_writer.write(img)
            print(info_show)

            self.ui.textBrowser.setText(info_show)

            show = cv2.resize(img, (640, 480))
            self.result = cv2.cvtColor(show, cv2.COLOR_BGR2RGB)
            showImage = QtGui.QImage(self.result.data, self.result.shape[1], self.result.shape[0],
                                     QtGui.QImage.Format_RGB888)
            self.ui.label.setPixmap(QtGui.QPixmap.fromImage(showImage))
            self.ui.label.setScaledContents(True)

        else:
            self.timer_video.stop()
            self.cap.release()
            self.vid_writer.release()
            self.ui.label.clear()

            self.ui.pushButton_video.setDisabled(False)
            self.ui.pushButton_img.setDisabled(False)
            self.ui.pushButton_camer.setDisabled(False)


    def button_video_stop(self):
        self.timer_video.blockSignals(False)

        if self.timer_video.isActive() == True and self.num_stop%2 == 1:
            self.ui.pushButton_stop.setText(u'Stop')
            self.num_stop = self.num_stop + 1
            self.timer_video.blockSignals(True)

        else:
            self.num_stop = self.num_stop + 1
            self.ui.pushButton_stop.setText(u'Continue')


    def finish_detect(self):

        self.cap.release()
        self.vid_writer.release()
        self.ui.label.clear()

        self.ui.pushButton_video.setDisabled(False)
        self.ui.pushButton_img.setDisabled(False)
        self.ui.pushButton_camer.setDisabled(False)


        if self.num_stop%2 == 0:
            print("Reset stop/begin!")
            self.ui.pushButton_stop.setText(u'stop/continue')
            self.num_stop = self.num_stop + 1
            self.timer_video.blockSignals(False)


if __name__ == '__main__':
    app = QtWidgets.QApplication(sys.argv)
    current_ui = UI_Logic_Window()
    current_ui.show()
    sys.exit(app.exec_())
```