Object detection for robots

Integration

Finally, we need to integrate our model with ROS. We will do so by editing the file `integration_activity.py`.

Simulation 💻

If you don't have a Jetson Nano Duckiebot, you can run this exercise locally with the simulator. This will run your code as a Pytorch model. This means that we rely on your host machine's CPU or GPU to run your model.

Hardware 🚙

If you have a Jetson Nano Duckiebot, you can run this exercise on your Duckiebot. This will convert your model to a TensorRT model, and run it your Jetson Nano's GPU.

What we are doing in this notebook

In both cases, we need to edit the ROS node to decide how we will use the detections. Should you call your model on every image from your camera? Only once every 4-5 frames, to preserve your CPU? You might also want to change your robot's behaviour depending on the size of the detection. If it is small, the object is probably far away, so there's no need to stop.

To get started, open the `integration_activity.py` file in the `packages/solution` directory.

You'll first need to input your information into the `DT_TOKEN` and `MODEL_NAME` functions. Follow the `TODO` markers in both functions to copy in your Duckietown token and the name of your uploaded model from the previous notebook.

Let us now compile our agent with the neural network model we trained and the behavior we implemented in the `integration_activity.py` file above.

You can build your agent by running the command,

```
dts code build
```

Then, you can use the following command to test your agent in simulation. The behavior of the driving agent here is to drive in a straight line until a detection triggers a stop. You will be able to control how detections can turn into stop signals by editing the `integration_activity.py` file. More on this later in the notebook.

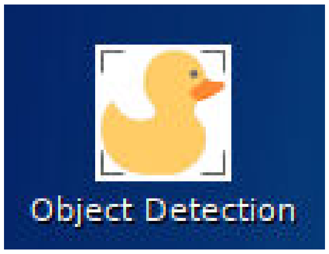
Let us run the simulated agent to see our neural network at work.

```
dts code workbench --sim
```

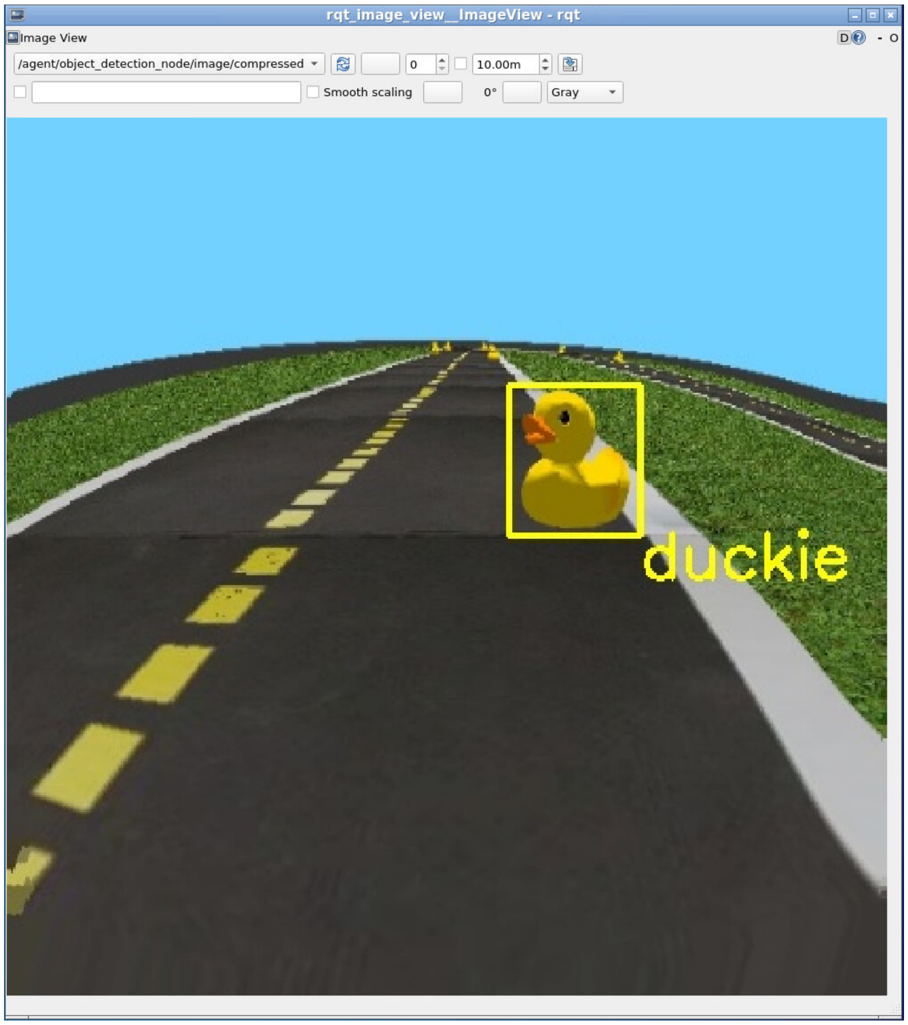
If you have a Duckiebot, run the following commands to test object detection in the real world:

```
dts code workbench -b <DUCKIEBOT_NAME>
```

In both cases, you will be given a VNC url in the terminal. Click on it to visualize in real-time the objects that the neural network we just trained detected in the camera image. In the VNC desktop that opens up, click on the "Object Detection" icon.



A stream of images from our agent's camera will appear, with the bounding boxes detected by our freshly trained neural network superimposed. An example is shown below.


rqt_image_view__ImageView - rqt
Image View
/agent/object_detection_node/image/compressed
0
10.00m
Smooth scaling
0°
Gray
duckie

Framerate

While object detection is useful, it is also very expensive, computationally.

One trick used to reduce said cost is to only call the full model infrequently. For example, one might call the model only a few times a second, which is very slow in computer timeframes, but relatively fast for the real world.

Of course, this varies from application to application. In very dynamic, fast robotic environments, clearly the model should be called more frequently.

You can fine-tune this yourself: edit the `NUMBER_FRAMES_SKIPPED` function in the `integration_activity.py` file to indicate the number of frames your robot should skip before calling its object detection model. A (default) value of `0` indicates that no frames are skipped (i.e., all frames are fed to the neural network for object detection), a value of `1` means that we are detecting objects every other frame, etc.

In real life, we would use a full neural network model to produce very accurate predictions, and then a less accurate model coupled with a Kalman filter (or other such estimation system) to "track" each prediction during the skipped frames.

For this exercises, we will limit ourselves to just skipping frames.

Filtering

Some of your predictions might not actually be useful. For example, in Duckietown, the trucks and busses are always parked on the side of the road. Your robot will never have to avoid or stop for them.

Cones can be in the road in some maps, but for this exercises, you can assume that there aren't any.

Filtering by class

For this reason, you probably want to remove all non-duckies from your predictions, since only duckies will be on the road. Update the `filter_by_classes` function to do this.

Filtering by score

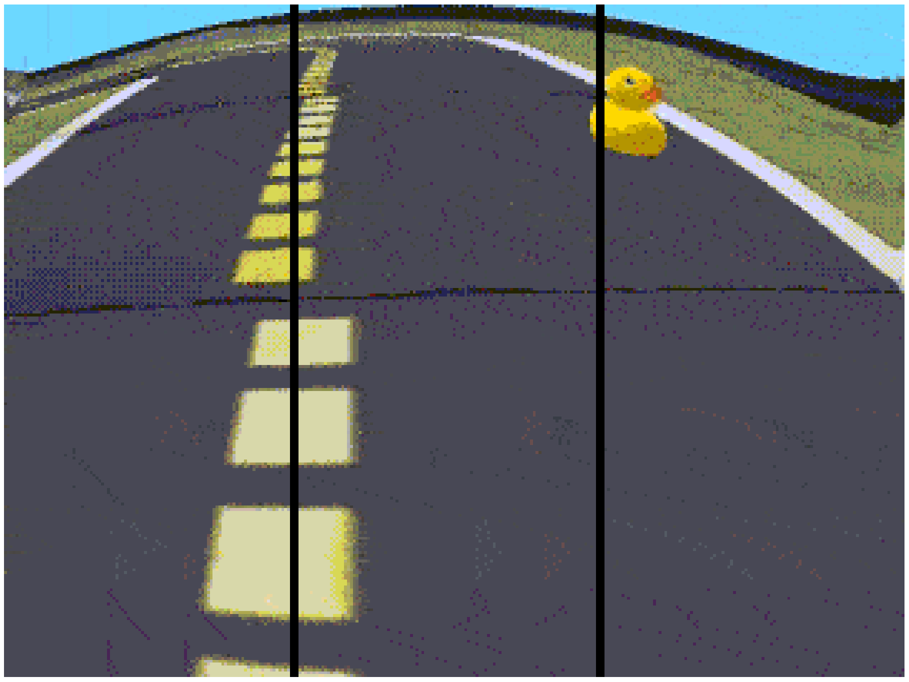
Depending on the model, you might also want to remove very unconfident detections.

Then again, your model might not be confident even for detections that are absolutely correct. You should experiment with the `filter_by_scores` function to find a value that works well for your model.

Filtering by bounding box

Finally, you should also evaluate what each detection *means* in terms of positionning.

If a bounding box is in the leftmost or rightmost thirds of the image, it might be the case that the object in not even on the road, and that your robot would be able to go by it without issue.



Also, if a bounding box's area is small, its object is likely far away. So there is no need to try to avoid the object or stop the robot: the robot still has a bit of driving to do before it reaches the object. So filtering out small detections might be a good idea too. Update the `filter_by_bboxes` function to play around with this.

Fine-tuning

In all of the functions above, there is not objective right answer. You should play with your functions and fine-tune their behaviours. Robotics is an iterative process!

Submission

You can test your agent locally with

```
$ dts code evaluate
```

And then finally submit with

```
$ dts code submit
```

And then check out how you did on the leaderboard.