# Learning-based Robotics Architectures using Duckietown

# Part A
# Object Detection Exercise

In this exercise you will train an object detection neural network. First, you will create your own training dataset with the Duckietown simulator. By using the segmentation maps that it outputs you will be able to label your dataset automatically. Then, you will adapt a pre-trained model to the task of detecting various classes of objects in the simulator. You will be graded on the quality of the dataset you made and the performance of your neural network model.

Contents

UNIT A-1

# Object detection

In this exercise you will train an object detection neural network. First, you will create your own training dataset with the Duckietown simulator. By using the segmentation maps that it outputs you will be able to label your dataset automatically. Then, you will adapt a pre-trained model to the task of detecting various classes of objects in the simulator. You will be graded on the quality of the dataset you made and the performance of your neural network model.

KNOWLEDGE AND ACTIVITY GRAPH

**Requires:** Some theory about machine learning

**Requires:** A proper laptop setup.

**Requires:** Some time for neural network training.

**Requires:** A pinch of patience.

**Results:** Get a feeling of what machine learning is.

Contents



## 1.1. Setup

**Note:** Setup a virtual environment! If you don't do this, your Python setup might get confused between the modified version of the simulator we will be using and the normal one that you might have previously installed globally.

We recommend using PyCharm, a Python IDE that includes support for `venv` from the get-go. PyCharm is free for students, so make sure you sign up with your university account.

Clone the template for this assignment.

Then, use the provided utility script to clone the special simulator for this homework (you might have to use `chmod +x` to make the script executable):

```
$ ./clone.sh
```

Finally, use your IDE of choice to install the `requirements.txt` that just got copied to the root of this directory into your `venv`. On PyCharm, simply press Alt-Enter on every

line of `requirements.txt` and select the option to install it (this might work only on recent PyCharm versions and requires that you enable the requirements plugin). You can also right-click in the `requirements.txt` file and select `Install All Packages`. Alternatively, run the following (make sure you are in the virtual environment you just created):

💻 `$ pip3 install -r requirements.txt`

## 1.2. Step 1: Investigation

What does an object detection dataset look like? What information do we need?

Try downloading the PennFudanPed dataset, a sample pedestrian detection dataset.

The first step of this exercise is simply understanding what's in that dataset. You'll notice that if you try opening the masks in that dataset, your computer will display a black image. That's because each segmented pedestrian's mask is a single digit and the image only has one channel, even though the mask was saved as a `.jpg`.

Try scaling the masks from 0 to 255, using something like `np.floor(mask / np.max(mask) * 255).astype(np.uint8)`. This will make the masks into something akin to a `.bmp`. Then, use OpenCv's `applyColorMap` feature on that to visualize the results. Try looking at the two `display` functions found in `utils.py` for inspiration.

This is all optional, of course. But we highly recommend trying it out, so that you can have an intuition for the type of images you should collect in the next step.

You'll also notice that the dataset doesn't include any bounding boxes. That's okay. For training with PennFudanPed, we have to compute them through numpy and OpenCV, just like we will on your own dataset.

Actually, for our own training, we won't need the masks! All we want are the bounding boxes. But PennFudanPed is a useful example, as it shows how we can extract bounding boxes from masks, something we will also do for our own dataset. To see how to do this, you may skip ahead to the tutorial linked in the Training section.

## 1.3. Step 2: Data collection

Now that we know what data we have to collect, we can start collecting it.

Do note that in this exercise, we don't want to differentiate the objects from one another: they will all have the same class. Our images will include duckies, busses, trucks, and cones.

We thus have five classes:

- 0: background
- 1: duckie
- 2: cone
- 3: truck
- 4: bus

To collect our data, we'll use the `segmented` flag in the simulator. Try running the `da-`

`ta_collection.py` file, which cycles between the segmented simulator and the normal one. Notice that, unfortunately, our duckie citizens are still novice in the field of computer vision, and they couldn't figure out how to remove the noise generated from their segmentation algorithm in the segmented images. That's why there's all this odd coloured "snow".

Notice that when we're in the segmented simulator, all the objects we're interested in have the exact same color, and the lighting and domain randomization are turned off. Just like the `data_collection.py` file does, we can also turn the segmentation back off for the same position of the agent. In other words, we can essentially produce two 100 identical images, save for the fact that one is segmented and the other is not.

Then, collect the dataset:

- We want as many images as reasonable. The more data you have, the better your model, but also, the longer your training time.
- We want to remove all non-`classes` pixels in the segmented images. You'll have to identify the white lines, the yellow lines, the stop lines, etc, and remove them from the masks. Do the same for the coloured "snow" that appears in the segmented images.
- We want to identify each class by the numbers mentioned above
- We also want the bounding boxes, and corresponding classes.

Your dataset must respect a certain format. The images must be 224x224x3 images. The boxes must be in `[xmin, ymin, xmax, ymax]` format. The labels must be an `np.array` indexed the same way as the boxes (so `labels[i]` is the label of `boxes[i]`).

We want to be able to read your `.npz`, so you *must* respect this format:

```
img = data["arr_0}"]
boxes = data["arr_1"]
classes = data["arr_2"]
```

Additionally, each `.npz` file must be identified by a number. So, if your dataset contains 1000 items, you'll have `npz` files ranging from `0.npz` to `999.npz`.

Do note that even though your dataset images have to be of size 224x224, you are allowed to feed smaller or bigger images to your model. If you wish to do so, simply resize the images at train/test/validation time.

**Hint:** You might want to take a look at the following OpenCV functions:

- `findContours`, and its `hierarchy` output which can be handy for filtering inner contours;
- `boundingRect`;
- `morphologyEx` with a suitable structuring element and morphological operation;

**Tip:** You might also want to make two separate datasets: one for training, and one for validation. Depending on your model, around 2000 samples for training should probably be more than enough.

**Evaluation:** We will manually look at part of your dataset and make sure that your bounding boxes match with the images.

## 1.4. Step 3: Model training

Now that we have our dataset, we will train on it. You may use PyTorch or TensorFlow; it doesn't really matter because we'll Dockerize your implementation. Note that the Tensorflow and PyTorch packages are not in `requirements.txt`. You'll have to install the library you want to use manually in your virtual environment.

The creators of this exercise do have a soft spot for Pytorch, so we'll use it as an example. Also some of the template code is setup for PyTorch so you might need to edit it in order to work for TensorFlow. Hence, unless you have a very strong preference for TensorFlow, we recommend you to stick with PyTorch.

This being ML, and ML being a booming field dominated by blogposts and online tutorials, it would be folly for us not to expect you to Google "how 2 obj detection pytorch". Let us save you some time. Here's the first result: pyTorch's object detection tutorial. We'll loosely follow that template.

First, define your `Dataset` class. Like in the link, for any given image index, it should provide:

- The bounding boxes for each class in each image (contrary to the tutorial, you calculated this earlier in the Data collection part of this exercise);
- The class labels for each bounding box;
- The normal, non-segmented image;
- An ID for the image (you should just use the index of the `.npz`).

Needless to say, each of the items must be index-dependent (the nth item of `boxes` must correspond to the nth item of `labels`).

We don't need the areas or the masks here: we'll change the model so that we only predict boxes and labels. Here's the model we will use instead of the tutorial's suggestion:

```
model = torchvision.models.detection.fasterrcnn_resnet50_fpn(pre-
trained=True)

# get number of input features for the classifier
in_features = model.roi_heads.box_predictor.cls_score.in_features
# replace the pre-trained head with a new one
model.roi_heads.box_predictor = FastRCNNPredictor(in_features, 5)
```

Then, you can use your `Dataset` class to train your model.

### 1) A note on the tutorial

Make sure to carefully read the tutorial. Blindly copying it won't directly work. The training data it expects is very specific, and you should make sure that you follow its structure exactly.

For example, the `PennFudanDataset` class does many pre-processing steps that you should have already performed in the data collection step. Hence, your dataset should already be (almost) ready for training.

Additionally, weirdly enough, the tutorial expects you to have some files that it does not link to.

Perhaps having a look (and a download) at these links might save you some time:

- engine.py
- coco_utils.py
- transforms.py

You can also safely remove the `evaluate` call that the tutorial uses, and it will save you the headache of installing most of the `coco_utils` and `coco_evaluate` dependencies.

### 2) Making sure your model does the right thing

You should probably write a way to visually evaluate the performance of your model.

Something like displaying the input image and overlaying the bounding boxes (colored by class) would be simple but very effective.

You should also carefully read `model.py`, as there are comments in it that describe the API your wrapper should respect.

### 3) Training hardware

But how should you actually train your model? If you have a recent-ish nVidia GPU, you can directly train on your computer. For reference, using a dataset with 2000 items, training on a GTX960 or a Quadro M1000M was very doable.

If you don't have a GPU, or if your GPU is too slow, you can still train and evaluate on your CPU. It is going to be slow but will work.

Alternatively, you can also use Google Colab. We included a `.ipynb` in the `model` directory. You can open it with Google Colab, upload the root of this exercise to your Google Drive, and the provided notebook will mount the folder from your drive into the Colab runtime, and then call your training script. To access the saved weights, simply download them from your Google Drive.

You can also improve the training speed by simplifying your model too and it might be easier to investigate that first.

### 4) Changing the model

The tutorial offers very good performance, but there are better options out there. You can essentially use any model you like here. However, make sure that it will work with our evaluation procedure. To ensure that, **do not change the interface of the `Wrapper` class** in the `model.py` source file!

We have also provided the setup that we will use for evaluating your models and a small dataset of 50 samples which you can use to ensure that your model will run when we evaluate it. Note that we will be using a different and bigger dataset, so make sure to not overfit!

Furthermore, feel free to replace the dataset in `eval/dataset` with a bigger one you've generated yourself should you wish to get a more accurate assessment of your model. Of course, do not use the same dataset for training and evaluation!

Apart from the `dataset` folder **do not change anything in the `eval` directory**! Should you do that, we don't guarantee that we will be able to evaluate your model anymore. The exact evaluate procedure is described in the next section.

Make sure that your model can be evaluated without a GPU, i.e. completely on a CPU. This involves checking if a GPU is available and initializing the model and the inputs in the right mode. In the provided template you can see some examples of the specific PyTorch functions you can use.

### 5) Evaluation

We will evaluate this section in two ways:

1. What is the accuracy of your model? Specifically, we will use mean average precision or mAP to evaluate your model, so you might want to optimize it for that metric.

2. Your complete model should be packages as a Docker image with all the dependencies and model weights included. We have provided a template Dockerfile in the root directory. This Docker image should be pushed to Dockerhub.

3. We will evaluate your model by using the same setting as in the `eval` directory but with a different dataset. We will first try to evaluate your model on a GPU by running: `make eval-gpu SUB={YOUR_IMAGE_NAME}` in the `eval` directory. However, if it does not work (incompatible hardware, wrong CUDA version, etc.) we will also attempt using a CPU alone. Hence, as mentioned above, make sure that your code runs without a GPU too. To evaluate without a GPU we will use the `make eval-cpu SUB={YOUR_IMAGE_NAME}` command. You can use the same two commands to verify that your image complies with the API we expect.

4. The above two commands will evaluate your code and will provide a final mAP value. You need to **score at least 80% mAP** for us to consider your model successful.

5. You will also have to provide the first 100 samples from the dataset you created.

# PART B
# Learning-based Control Exercise

In this exercise you will implement an LQR controller for your Duckiebot using the Duckietown simulator. First, you will collect Duckiebot data (states and control inputs) of the robot performing standard lane following. Second, you will train a machine learning model with the data collected, and lastly, you will use the model to implement an LQR controller and comment on your performance and implementation.

Contents

UNIT B-1

# Learning-based Control - Data Collection

KNOWLEDGE AND ACTIVITY GRAPH

Requires: Docker Basics (unknown ref duckietown-robotics-development/docker-basics)

warning (1 of 1) index

warning

```
I will ignore this because it is an external link.

 > I do not know what is indicated by the link
'#duckietown-robotics-development/docker-basics'.
```

Location not known more precisely.

Created by function `n/a` in module `n/a`.

Requires: Knowledge about machine learning

Requires: Some knowledge of control theory

Results: Implement an LQR Controller using learning for your Duckiebot.

**Contents**



## 1.1. Overview of the task

For this task, we will be using the Duckietown Gym to run standard lane following in simulation and obtain data of the Duckiebot model. We will collect the information about the state $\vec{x} = [d \quad \varphi]^\intercal$, and the control input $u$, where $x \in \mathcal{R}^2$ and $u \in \mathcal{R}$.

- $d$ is the distance of the center of the Duckiebot's axle to the center of the right lane.
- $\varphi$ is the angle (rads) from centerline to heading of the Duckiebot.
- $u$ is the steering input (in radians per second).

## 1.2. Duckietown Gym - Duckietown Exercises Notebook

We will be working in the Duckietown Gym. Gym-Duckietown is a simulator for the Duckietown Universe, written in pure Python/OpenGL (Pyglet). It places your agent, a Duckiebot, inside of an instance of a Duckietown: a loop of roads with turns, intersections, obstacles, Duckie pedestrians, and other Duckiebots. If not yet familiar, please

read the docs directly on the Gym-Duckietown to get a better understanding about working with the simulator.

We will run the exercise directly from a Jupyter Notebook. To run, follow the next steps:

```
$ git clone --recursive https://github.com/duckietown-ethz/lqr-exercise.git

$ cd lqr-exercise

$ pip3 install -r requirements.txt

$ dts exercises notebooks
```

This is where you will develop your solution. You will find a Jupyter-notebook for model training and sample collection. In addition, the `lraClass.py` file has been included to facilitate loading and saving your data.

Troubleshooting: If you forgot to do `--recursive` you can type `git submodule init` and then `git submodule update` from the exercise root folder.

Troubleshooting: You might need to install the image display package ffmpeg, you can do this by typing `sudo apt-get install ffmpeg`.

## 1.3. Data collection

### 1) Collecting the data using default Controller

The first task is to collect the state and control input data. For this, you will need to create an Gym instance of the Duckietown Environment. An example of this setup procedure is shown below. The function `env.render()` displays a graphical representation of the environment to your screen.

```
from simulator.src.gym_duckietown.simulator import Simulator
#   Create the environment

env = Simulator(
    map_name = "udem1",
    domain_rand = False,
    draw_bbox = False
)

obs = env.reset()
env.render()
```

You can use the default controller `basic_control.py` to make your Duckiebot move. In the while loop, you will need to do two things:

1. Extract the steering input (in radians per second)
2. Collect the data.

Run your simulation until it finished or crashes, and then save the data to a csv file.

Tip: For data collection, you can use the LRA helper function `collectData(u,x)` which takes inputs a scalar $u$ and a vector $\vec{x}$.

Tip: For saving your data, you can use the LRA helper function `saveData("filepath.csv")`.

You should now have a `.csv` file containing your training data. An example of what this file should look like is shown below:

```
d,phi,u
-0.1618443443591879,-0.298587790613436,-1.6330326457217237
-0.16389157947457456,-0.29663545184459117,-1.632756126910639
-0.16590270172174604,-0.29818014534655817,-1.6122521921396433
-0.17345815224603006,-0.25618446121623173,-1.5169998357130914
-0.17518546625467168,-0.2396585301709049,-1.4624035526137578
-0.1767672947637351,-0.21713463020459525,-1.3854049199878498
...
...
```

### 2) Collecting the data with random control signals

Repeat the process you just did, but this time use random control signals that are uncorrelated to the states. Run the simulation and record the states, and control input. Save the file for your random control input as a separate `.csv` file.

A successful run will output the video of your Duckiebot moving in the simulation. A sample output can be seen below:

Figure 1.1. Example video from data collection process with PID control

## 1.4. Model Training

The next step is to use the data you collected to develop a linear model of the Duckiebot. The model can be written as follows:

$$\vec{X}_{t+1} = A\vec{X}_t + B\vec{u}_t$$

Where, * $A \in \mathcal{R}^{2x2}$ * $B \in \mathcal{R}^{2x1}$

The goal is to find the matrices $\boldsymbol{A}$ and $\boldsymbol{B}$.

### 1) Data Cleaning

Not all the collected data might be relevant, it is your task now to determine what data to include and why.

The data can be easily loaded using the LRA helper function `loadData("data.csv")`.

```
lra = LRA2_HELPER()
data = lra.loadData("model_data.csv")
```

Hint: the data is imported as a pandas DataFrame. This can easily be sliced, truncated, sorted, and allows for other operations. You should look at the `.iloc[]` and `.loc[]` methods to facilitate data cleaning and pre-processing.

> **Note:** Note that an important factor is that the control signal should be bound by a magnitude of 1.
>
> $$|u| \leq 1$$

### 2) Model training

You are free to use any model you want for training. The goal is to use regression to find $\boldsymbol{A}$ and $\boldsymbol{B}$. One way to do this is to use `sklearn.linear_model` library to initialize a model.

In `sklearn` models follow the convention of X,Y, where X represents the training data, and Y the outputs. The model you are trying to predict is shown below: Keep in mind that we are fitting a linear model.

$$[d_{t+1}\ \varphi_{t+1}] = [A \quad |B]\,[d_t\ \varphi_t\ u]$$

Deliverable: Submit your A and B matrices for the data collected using the PID controller.

Deliverable: Submit your A and B matrices for the data collected using random control signals.

Deliverable: What are the advantages and disadvantages of using the data collected using the PID controller?

Deliverable: What are the advantages and disadvantages of using the data collected using the random control signals?

Deliverable: Is all the data collected useful? Explain why or why not. In addition, elaborate on any type of data cleaning you might have performed.

## 1.5. Linear Quadratic Regulator - Control

Now that you have learned the model of the Duckiebot using machine learning, the objective is to implement a controller to test how accurate the model prediction is. To do this you will use a Linear Quadratic Regulator (LQR).

The LQR formulation is shown below:

$$J = \int_0^{\inf} (\vec{x}^T Q \vec{x} + \vec{u}^T R \vec{u})\, dt$$

The LQR is an optimization problem that penalizes the state and control inputs according to the matrices $Q$ and $R$, respectively. You will need to solve the discrete Algebraic Riccati Equation

The solution of the ARE can be used to obtain the gains. Compute the gain matrix K and implement it in your control. Run the simulator again, and record your screen for your Duckiebot running on your custom model-based-learning controller.

Deliverable: A short video (~5-10 sec) of your simulated Duckiebot running on your LQR controller.

Deliverable: Your Q and R matrices

Deliverable: Your K matrix

Deliverable: Question: How does the performance of your controller compare to the controller you used to collect data initially?

Deliverable: Question: Are you penalizing control input, why?

Deliverable: Question: Are you penalizing the states, why?