The AI Driving Olympics

Contents

Part A
Introduction

Contents

For a detailed description of the scientific objectives and outcomes please see our paper about the AI-DO at NeurIPS.

1) Quick links

There are different challenges, different computational resource regimes and different performance categories in this competition.

- Get started with your code submission

2) AI-DO 6 Urban League Challenges

- Lane following (LF)
- Lane following with vehicles (LFV)
- Lane following with intersections (LFVI)
- Lane following with multiple vehicles and intersections where state information is given (LFVI–multi-stateful)

3) Computational resources

- Purist option - RaspberryPi 3B+ (discontinued from AI-DO 6)
- Purist option - Jetson Nano (2 GB and 4 GB version)
- Remote option

Note that during the AI-DO 6 finals, the submissions will be run remotely.

4) Evaluation metrics

For details about the evaluation metrics please refer to the performance metrics

UNIT A-1

The AI Driving Olympics

The AI Driving Olympics (AI-DO) is a set of competitions with the objective of evaluating the state of the art in machine learning and artificial intelligence for mobile robotics.

For a detailed description of the scientific objectives and outcomes please see our paper about the AI-DO at NeurIPS.

Figure 1.1. The AI Driving Olympics at ICRA 2020

Contents

1.1. History

- **AI-DO 1** was in conjunction with **NeurIPS 2018**.
- **AI-DO 2** was in conjunction with **ICRA 2019**.
- **AI-DO 3** was in conjunction with **NeurIPS 2019**.
- **AI-DO 4** was supposed to be in conjunction with **ICRA 2020**, but was canceled due to COVID-19.
- **AI-DO 5** was in conjunction with **NeurIPS 2020**.
- **AI-DO 6** is in conjunction with **NeurIPS 2021**.

Figure 1.2. Where it all started: AI-DO 1 at NeurIPS 2018 in Montreal.

1.2. Leagues

There are currently three leagues in the AI Driving Olympics.

The **Urban League** is based on the Duckietown platform, and includes a series of tasks of increasing complexity. For each task, we provide tools for competitors to use in the form of simulators, logs, code templates, baseline implementations and low-cost access to robotic hardware. We evaluate submissions in simulation online, on standardized hardware environments, and finally at the competition event.

Participants will not need to be physically present at any stage of the competition — they will just need to send their source code.
There will be qualifying rounds in simulation, similar to recent DARPA Robotics Challenges, and, for evaluation, we make available the use of “Duckietown Autolabs (unknown ref opmanual_autolab/book)

warning next (1 of 18) index
warning

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

 > I do not know what is indicated by the link '#opman-
ual_autolab/book'.
```

Location not known more precisely.

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

” which are facilities that allow remote experimentation in a reproducible setting.
See the leaderboards and many other things at the challenges site.

The **Advanced Perception League** is organized by Motional (ex nuTonomy, Aptiv Mobility).

All information about the Advanced Perception League is at nuScenes.org.

The **Racing League** is organized by the AWS Deepracer team. All information about the racing league is available on aicrowd.com.

1.3. What’s new in the Urban League in AI-DO 6

There have been cool new improvements for the 6th edition of the AI-DO Urban League:

- The challenges are now compatible the new DB21 Duckiebots that have Jetson Nanos with GPUs and were used for the Self-Driving Cars with Duckietown MOOC on EdX.

1.4. How to use this documentation

If you would like to compete in the AI-DO Urban League, you will want to:

- Read the brief introduction to the competition.
- Find the challenge that you would like to try.
- Get started and make a submission.

At this point you are all set up: your environment is operational, and you can make a submission. But you should want to make your submission perform better than the provided baselines.

To do this the following tools might prove useful:

- The AIDO API so that your workflow is efficient using the available tools.
- The reference algorithms where we have implemented some different approaches to approach the challenges.

How to get help

If you are stuck try one of the following things:

- Look through the contents of this documentation using the links on the left. Note that the “Parts” have many “Chapters” that you can see when you click on the Part title,
- Join our slack community,
- Look on the Duckietown Stack Overflow to see if someone already answered your question (you can ask to be invited in the Slack channel #help-accounts)
- If you are sure you actually found a bug, file a Github issue in the appropriate repo.

How to cite

If you use the AI-DO platform in your work and want to cite it please use:

```
@article{zilly2019ai,
  title={The AI Driving Olympics at NeurIPS 2018},
  author={Julian Zilly and Jacopo Tani and Breandan Considine and
Bhairav Mehta and Andrea F. Daniele and Manfred Diaz and Gianmarco
Bernasconi and Claudio Ruch and Jan Hakenberg and Florian Golemo and A.
Kirsten Bowser and Matthew R. Walter and Ruslan Hristov and Sunil
Mallya and Emilio Frazzoli and Andrea Censi and Liam Paull},
  journal={arXiv preprint arXiv:1903.02503},
  year={2019}
}
```

If you use the Duckietown platform in your work and want to cite it please use:

```
@INPROCEEDINGS{PaullICRA2017,
    author={Paull, Liam and Tani, Jacopo and Ahn, Heejin and Alonso-Mo-
ra, Javier and Carlone, Luca and Cap, Michal and Chen, Yu Fan and Choi,
Changhyun and Dusek, Jeff and Fang, Yajun and Hoehener, Daniel and Liu,
Shih-Yuan and Novitzky, Michael and Okuyama, Igor Franzoni and Pazis,
Jason and Rosman, Guy and Varricchio, Valerio and Wang, Hsueh-Cheng and
Yershov, Dmitry and Zhao, Hang and Benjamin, Michael and Carr, Christo-
pher and Zuber, Maria and Karaman, Sertac and Frazzoli, Emilio and Del
Vecchio, Domitilla and Rus, Daniela and How, Jonathan and Leonard, John
and Censi, Andrea},
    booktitle={2017 IEEE International Conference on Robotics and Au-
tomation (ICRA)}, title={Duckietown: An open, inexpensive and flexible
platform for autonomy education and research},
    year={2017},
    volume=,
    number=,
    pages={1497-1504},
```

UNIT A-2
The Duckietown Platform

The Duckietown platform has many components.

This section focuses on the physical platform used for the embodied robotic challenges.

For examples of Duckiebot driving see a set of demo videos of Duckiebots driving in Duckietown (unknown ref opmanual_duckiebot/demos)

previous **warning** next (2 of 18) index
warning

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

 > I do not know what is indicated by the link '#opman-
ual_duckiebot/demos'.
```

Location not known more precisely.

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

.

The actual embodied challenges will be described in more detail in LF, LFV, LFI.

> Note: the sequence of the challenges was chosen to gradually increase the difficulty, by extending previous challenge solutions to more general situations. We recommend you tackle the challenges in this same order.

Contents

2.1. The Duckietown Platform

There are three main parts of the platform with which you will interact:

1. **Simulation and training** environment, which allows testing in simulation before trying on the real robots.
2. **Duckietown Autolabs** in which to try the code in controlled and reproducible conditions.
3. **Physical Duckietown platform**: miniature autonomous vehicles and smart-cities in which the vehicles drive. The Duckiebots (unknown ref opmanual_duckiebot/duckiebot-configurations)

previous **warning** next (3 of 18) index
warning

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

 > I do not know what is indicated by the link '#opman-
ual_duckiebot/duckiebot-configurations'.
```

Location not known more precisely.
Created by function `n/a` in module `n/a`.

(robot hardware) and Duckietown (environment) are rigorously specified (unknown ref opmanual_duckietown/dt-ops-appearance-specifications)

previous **warning** next (4 of 18) index
warning

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

 > I do not know what is indicated by the link '#opman-
ual_duckietown/dt-ops-appearance-specifications'.
```

Location not known more precisely.
Created by function `n/a` in module `n/a`.

, which makes the development extremely repeatable. If you have a Duckiebot you can refer to the Duckiebot operational manual (unknown ref opmanual_duckiebot/book)

previous **warning** next (5 of 18) index
warning

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

 > I do not know what is indicated by the link '#opman-
ual_duckiebot/book'.
```

Location not known more precisely.
Created by function `n/a` in module `n/a`.

for step-by-step instructions on how to assemble, maintain, calibrate and operate your robot. If you would like to acquire a Duckiebot please go to the Duckietown project store.

The Duckiebots officially supported for AI-DO 6 (2021) are the `DB21` Duckiebots. We recommend you build your Duckietowns according to the specifications, too. The necessary materials can be sourced locally pretty much globally - but if you want compliant “one-click” AI-DO kits for each challenge you can get them from here:

- `LF` AI-DO 6 hardware kit
- `LFV` AI-DO 6 hardware kit
- `LFI` AI-DO 6 hardware kit

For any questions regarding Duckietown hardware you can reach out to `hardware@duckietown.com`.

2.2. Duckiebots and Duckietowns

We briefly describe the physical Duckietown platform, which comprises autonomous vehicles (*Duckiebots*) and a customizable model urban environment (*Duckietown*).

1) The Duckiebot

Duckiebots are designed with the objectives of affordability, modularity and ease of construction. They are equipped with: a front viewing camera with 160 degrees fish-eye lens capable of streaming 0 0 resolution images reliably at 30 fps, and wheel encoders on the motors. `DB21` Duckiebots are equipped with IMUs and front facing time of flight sensors too.

Actuation is provided through two DC motors that independently drive the front wheels (differential drive configuration), while the rear end of the Duckiebot is equipped with a passive omnidirectional wheel.

All the *computation* is done onboard on a: - `DB19`: Raspberry Pi 3B+ computer, - `DB21`: Jetson Nano 2 GB (`DB21M`) or Jetson Nano 4 GB (`DB21J`).

Power is provided by a 0000 mAh Duckiebattery (unknown ref opmanual_duckiebot/db-opmanual-preliminaries-electronics)

previous **warning** next (6 of 18) index
warning

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

 > I do not know what is indicated by the link '#opman-
ual_duckiebot/db-opmanual-preliminaries-electronics'.
```

Location not known more precisely.

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

which provides several hours of operation.

2) The Duckietown

Duckietowns are modular, structured environments built on two layers: the *road* and the *signal* layers (Figure 2.2). Detailed specifications can be found here (unknown ref opmanual_duckietown/dt-ops-appearance-specifications)

previous **warning** next (7 of 18) index
warning

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

 > I do not know what is indicated by the link '#opman-
ual_duckietown/dt-ops-appearance-specifications'.
```

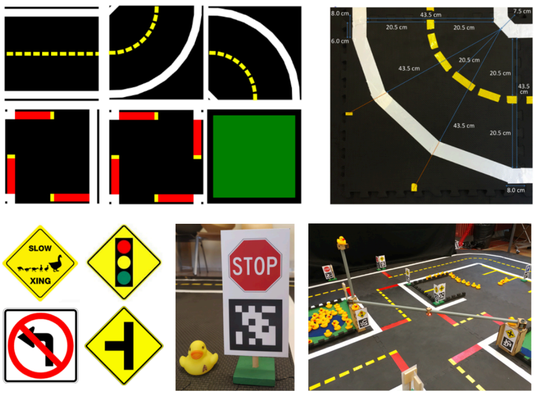
Location not known more precisely.

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

.

There are six well-defined *road segments*: straight, left and right 90 deg turns, 3-way intersection, 4-way intersection, and empty tile. Each is built on individual tiles, and their interlocking enables customizable city sizes and topographies. The appearance specifications detail the color and size of the lines as well as the geometry of the roads.

The signal layer comprises street signs and traffic lights. *Street signs* enable global localization (knowing where they are within a predefined map) of Duckiebots in the city and interpretation of intersection topologies. They are defined as the union of an AprilTag [1] in addition to the typical road sign symbol. Their size, height and relative positioning with respect to the road are specified. Many signs are supported, including intersection type (3- or 4-way), stop signs, road names, and pedestrian crossings.



The Duckietown environment is rigorously defined at road and signal level. When the appearance specifications are met, Duckiebots are guaranteed to navigate cities of any topology.

Figure 2.2

3) Simulation

We provide a cloud simulation environment for training.

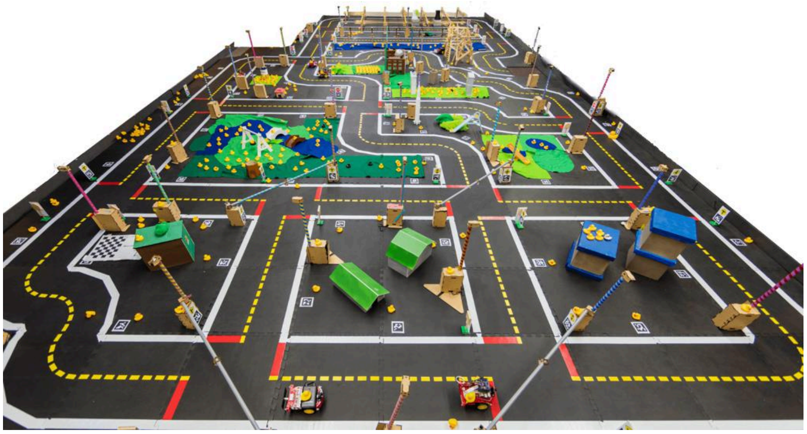
In a way similar to the last DARPA Robotics Challenge, we use the simulation as a first screening of the participant's submissions. It will be necessary for the submitted agent code to run in simulation and be sufficiently performant to gain access to the Autolabs.

Simulation environments for each of the individual challenges are provided as Docker containers with clearly specified APIs. The baseline solutions for each challenge is provided as separate containers. When both containers (the simulation and corresponding solution) are loaded and configured correctly, the simulation will effectively replace the real robot(s). A proposed solution can be uploaded to our cloud servers, at which point it will be automatically run against our pristine version of the simulation environment (on a cluster) and a score will be assigned and returned to the participant.

Examples of the simulators provided are shown on the Duckietown Challenges server. E.g., here is a LF evaluated submission example from AI-DO 5.

This simulator is also integrated with the OpenAI Gym environment for reinforcement learning agent training. An API for designing reward functions or tweaking domain randomization will be provided.

4) Duckietown Autolabs



The Duckietown Autolab at ETH Zürich

Figure 2.4

The idea of an Autolab is inspired by Georgia Tech's Robotarium (contraction of *robot* and *aquarium*) [2].

The use of an Autolab has two main advantages:

1. Convenience: It allows convenient access to a complete robot setup.

2. Reproducibility: It allows for multiple people to run the experiments in repeatable controlled conditions.

You can find detailed information on Duckietown Autolabs in our paper: Integrated Benchmarking and Design for Reproducible and Accessible Evaluation of Robotic Agents.

If you would like to cite Duckietown Autolabs, please use:

```
@INPROCEEDINGS{tani2020duckienet,
  author={Tani, Jacopo and Daniele, Andrea F. and Bernasconi, Gianmarco
and Camus, Amaury and Petrov, Aleksandar and Courchesne, Anthony and
Mehta, Bhairav and Suri, Rohit and Zaluska, Tomasz and Walter, Matthew
R. and Frazzoli, Emilio and Paull, Liam and Censi, Andrea},
  booktitle={2020 IEEE/RSJ International Conference on Intelligent Ro-
bots and Systems (IROS)},
  title={Integrated Benchmarking and Design for Reproducible and Acces-
sible Evaluation of Robotic Agents},
  year={2020},
  volume=,
  number=,
  pages={6229-6236},
  doi={10.1109/IROS45743.2020.9341677}}
```

For the competition we will several options for computational power.

1. The "purist" computational substrate option: where processing is done onboard Duckiebots.

2. The images are streamed to a base-station with a powerful GPU. This will increase computational power but also increase the latency in the control loop.

Part B
The Challenges

This section precisely defines the general rules and performance metrics and explains AI-DO Urban league challenges.

Contents

UNIT B-1
General rules

Contents

1.1. Protocol

1) Deployment technique

We use Docker containers to package, deploy, and run the applications on the physical Duckietown platform as well as on the cloud for simulation. Base Docker container images are provided and distributed via Docker HUB.

A **challenges server** is used to collect and queue all submitted agents. The **simulation evaluations** execute each queued agents as they become available. Submissions that pass the simulation environment will be queued for execution in the Autolab.

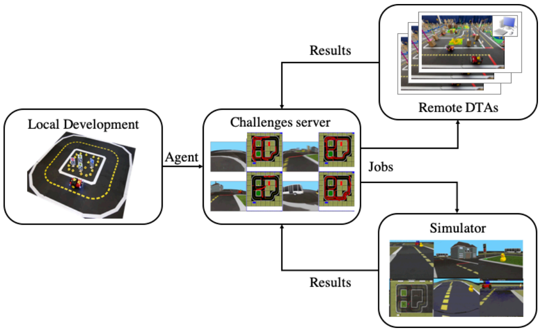


Figure 1.1. The AI-DO evaluations workflow supports local and remote development, in simulation and on hardware.

For validation of submitted code and evaluation the competition finals a surprise environment will be employed. This is to discourage over-fitting to any particular Duckietown configuration.

2) Submission of entries

Participants can submit their code in the form of a docker container to a challenge. Templates are provided for creating the container image in a conforming way.

The system will schedule to run the submitted robot agent on the cloud on the challenges selected by the user, and, if simulations pass, in the Autolabs.

Participants can submit entries as many times as they would like, which will be processed on a best effort basis. Access control and prioritization policies are in place to provide equal opportunities to all participants and prevent monopolization of the computational and physical resources available.

Participants are required to open source their solutions source code. If auxiliary training data are used to train the models, that data must be made available.

Submitted code will be evaluated in simulation and if sufficient on physical Autolabs. Scores and logs generated with submitted code are made available on the challenges server.

Simulation code is available as open source for everybody to use on computers that they control. The baselines interact with the simulator through a standardized interfaces that mimics the interface with the real robot.

3) Autolab test and validation

When an experiment is run in a **training/testing** Autolab, the participants receive, in addition to the score, detailed feedback, including logs, telemetry, videos, etc. The sensory data generated by the robots is continuously recorded and becomes available to the entire community.

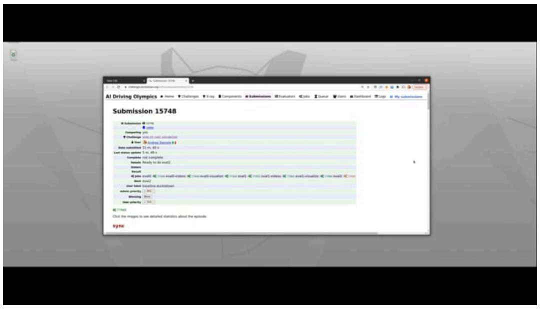


Figure 1.2. Autolab LF-challenge evaluation demo.

When an experiment is run in a **validation** Autolab, the only output to the user is the test score and minimal statistics (number of collisions, number of rule violations, etc.). Here are some examples.

4) Leaderboards

After each run in simulation and in Autolabs, the participants can see the metrics statistics on the competition scoring website. Extended leaderboards are made available for each challenge.

1.2. Eligibility

Employees and affiliates of organizing and sponsoring organizations are ineligible from participation in the competition, but they are welcome to submit baseline solutions that will be reported in a special leaderboard.

Students of organizing institutions (ETH Zürich, University of Montreal, and TTIC), are eligible to participate in the competition as part of coursework, if they do not work in the organization of the competition.

1.3. Intellectual property

Participants of AI-DO are required to provide the source code / data / learning models of their submission to the organizers before the finals (so that we can check for their regularity.)

Winners of AI-DO are required to make their submission open source so that it can be reused later in the next challenges.

UNIT B-2
Performance metrics

Measuring performance in robotics is less clear cut and more multidimensional than traditionally encountered in machine learning settings. Nonetheless, to achieve reliable performance estimates we assess submitted code on several *episodes* with different initial settings and compute statistics on the outcomes. We denote $\mathcal{J}$ to be an objective or cost function to optimize, which we evaluate for every experiment. In the following formalization, objectives are assumed to be minimized.

In the following we summarize the objectives used to quantify how well an embodied task is completed. We will produce scores in three different categories.

Contents

2.1. Performance criteria (P)

As a performance indicator for both the “lane following task” and the “lane following task with other dynamic vehicles”, we choose the integrated speed $v(t)$ along the road (not perpendicular to it) over time of the Duckiebot. This measures the moved distance along the road per episode, where we fix the time length of an episode. This encourages both faster driving as well as algorithms with lower latency. An *episode* is used to mean running the code from a particular initial configuration.

$$\mathcal{J}_{P-LF(V)}(t) = \int_0^t -v(t)dt$$

The integral of speed is defined over the traveled distance of an episode up to time $t = T_{eps}$, where T_{eps} is the length of an episode.

The way we measure this is in units of “tiles traveled”:

$$\mathcal{J}_{P-LF(V)}(t) = \#\ \text{of tiles traveled}$$

2.2. Traffic law objective (T)

The following shows rule objectives the Duckiebots are supposed to abide by within Duckietown. These penalties hold for the embodied tasks (LF, LFV).

1) Major infractions

This objective means to penalize “illegal” driving behavior. As a cover for many undesired behaviors, we count the median time spent oustide of the drivable zones. This also covers the example of driving in the wrong lane.

Metric: The median of the time spent outside of the drivable zones.

$$\mathcal{J}_{T-LF/LFV} = \textbf{median}(\{t_{outside}\}),$$

where $\{t_{outside}\}$ is the list of accumulated time outside of drivable zones per episode.

2.3. Comfort objective (C)

In the single robot setting, we encourage "comfortable" driving solutions. We therefore penalize large angular deviations from the forward lane direction to achieve smoother driving. This is quantified through changes in Duckiebot angular orientation $\theta_{bot}(t)$ with respect to the lane driving direction.

Lateral deviation:

For better driving behavior we measure the median per episode lateral deviation from the right lane center line.

$$\mathcal{J}_{C-LF/LFV}(t) = \textbf{median}(\{d_{outside}\}),$$

where $\{d_{outside}\}$ is the sequence of lateral distances from the center line.

UNIT B-3
Challenge **LF**

The first challenge of the *AI Driving Olympics* is "lane following" (`LF`).

In this challenge, we ask participants to submit code allowing the Duckiebot to drive on the right-hand side of the street within Duckietown without a specific goal point. Duckiebots will drive through the Duckietown and will be judged on how fast they drive, how well they follow the rules and how smooth or "comfortable" their driving is.

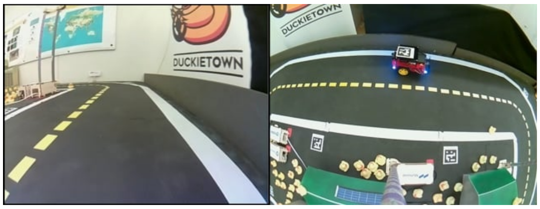


Figure 3.1. Lane following example submission.

A description of the specific rules is provided.

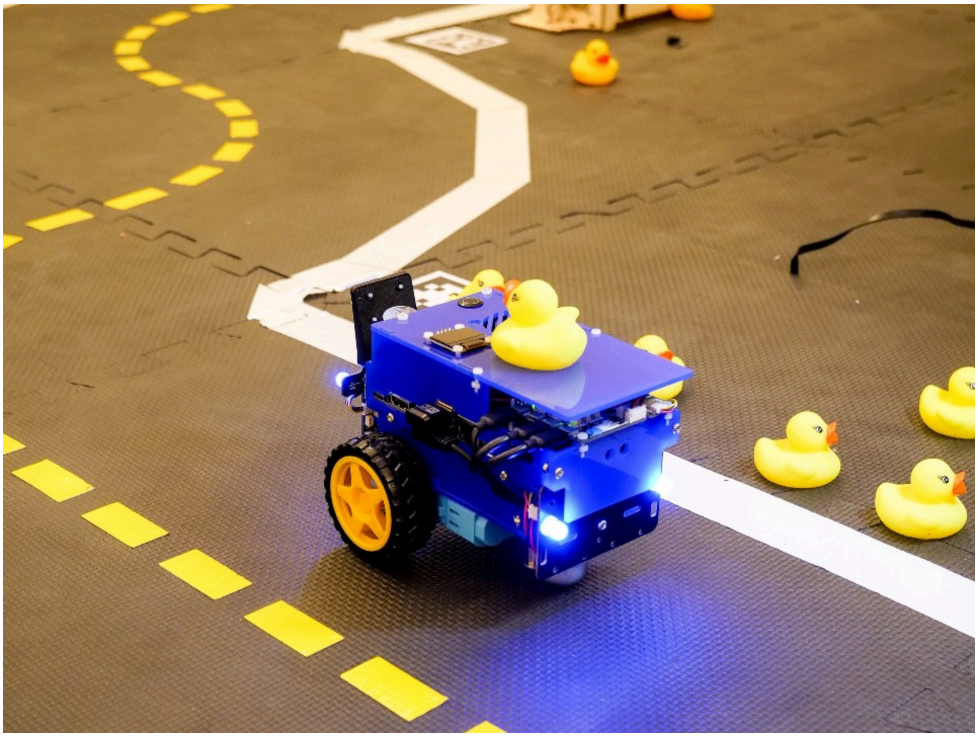


Figure 3.2. A Duckiebot following a lane.

The challenge is designed in a way that allows for a completely *reactive* algorithm design, i.e., to accomplish the challenge it is not strictly necessary to keep past observations in memory.

In particular intersections will not be part of maps for this challenge.

3.1. Templates and Baselines

To get started, try one of the existing templates, which are minimal setups that do random things but are functions, or the baselines which are instantiations of the templates that implement some algorithms, but not in an optimal way. Additionally, many of the past AI-DO winners are in the baseline solutions.

3.2. **LF** in Simulation

The current versions of the lane following simulation challenges are `aido-LF-sim-testing` and `aido-LF-sim-validation`. These two challenges are identical except for the output that you are allowed to see. In the case of `testing` you will be able to see performance of your agent (Figure 3.3) and you will be able to download the logs and artifacts.

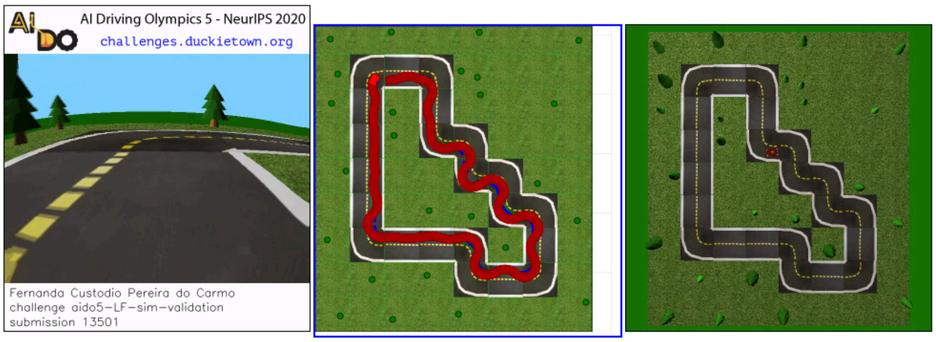

AI DO
AI Driving Olympics 5 - NeurIPS 2020
challenges.duckietown.org
Fernanda Custodio Pereira do Carmo
challenge aido5-LF-sim-validation
submission 13501

Figure 3.3. Visual output for a LF submission

1) `aido-LF-sim-testing` Details

- Challenge overview
- Leaderboard
- All submissions

The details for “experiment manager”, “simulator”, and “scenario maker” parameters may be of interest and are available here (Under “Details”).

2) `aido-LF-sim-validation` Details

- Challenge overview
- Leaderboard
- All submissions

3.3. **LF** in the Duckietown Autolab

The current version of the lane following real robot challenge is `aido-LF-real-vali-`

`dation`.

Note that to test the performance of your agent on the real robot yourself, you can follow the instructions to run your agent on your Duckiebot

1) `aido-LF-real-validation` Details

- Challenge overview
- Leaderboard
- All submissions

UNIT B-4
Challenge **LFV**

The second challenge of the *AI Driving Olympics* is “lane following with dynamic vehicles” (`LFV`). This challenge is an extension of Challenge `LF` to include additional rules of the road and other moving vehicles and static obstacles.

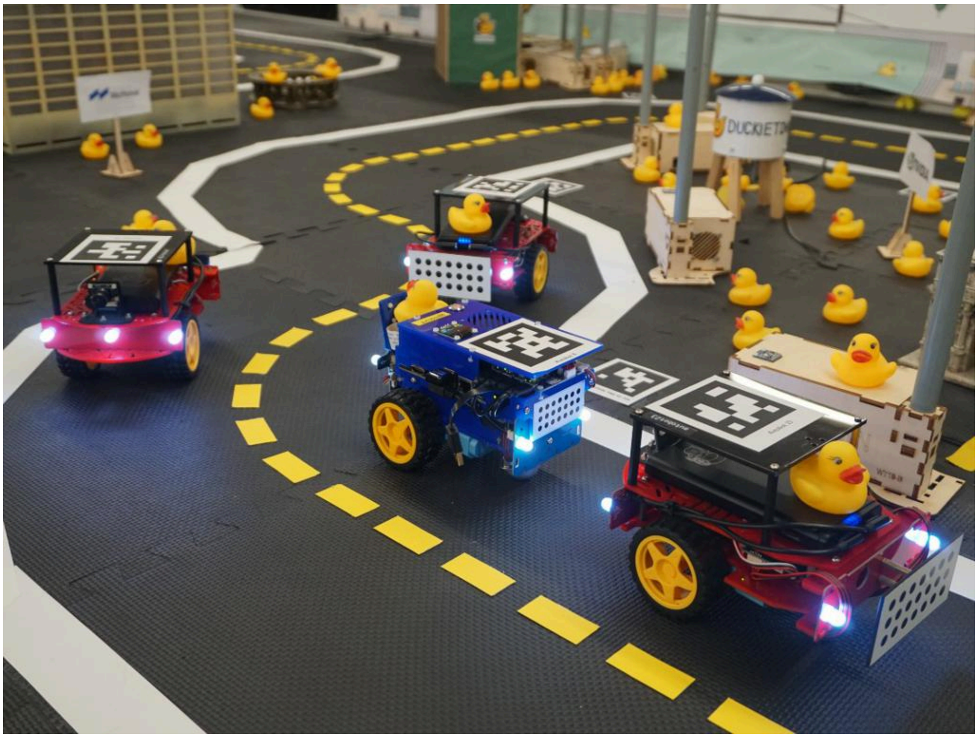


Figure 4.1. A Duckiebot doing lane following with other vehicles.

Again we ask participants to submit code allowing the Duckiebot to drive on the right-hand side of the street within Duckietown. Due to interactions with other Duckiebots, a successful solution will likely not be completely *reactive*.

4.1. **LFV** in Simulation

The current versions of the lane following with vehicles in simulation are `aido2-LFV-sim-testing` and `aido2-LF-sim-validation`. These two challenges are identical except for the output that you are allowed to see. In the case of `testing` you will be able to see performance of your agent (Figure 4.2) and you will be able to download the logs and artifacts.

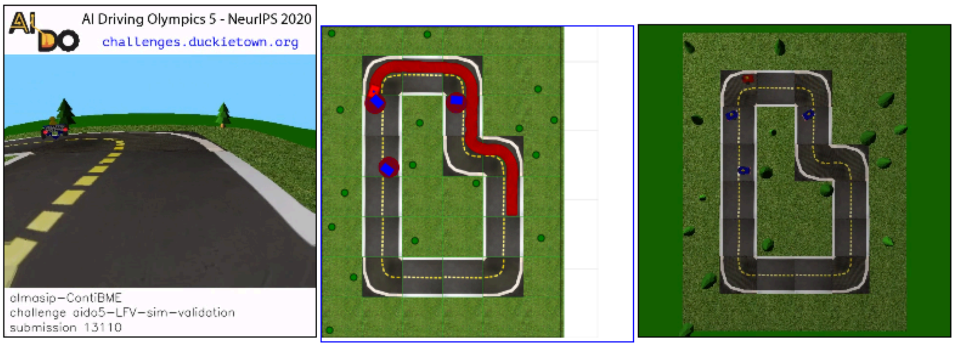

AI DO
AI Driving Olympics 5 - NeurIPS 2020
challenges.duckietown.org
almasip–ContiBME
challenge aido5–LFV–sim–validation
submission 13110

Figure 4.2. Visual output for a LFV submission.

4.2. Templates and Baselines

To get started, try one of the existing templates, which are minimal setups that do random things but are functions, or the baselines which are instantiations of the templates that implement some algorithms, but probably not in an optimal way. Many of the past AI-DO winners are in the baseline solutions.

1) **`aido-LFV-sim-testing`** Details

- Challenge overview
- Leaderboard
- All submissions

The details for "experiment manager", "simulator", and "scenario maker" parameters may be of interest and are available here (Under "Details").

2) **`aido-LFV-sim-validation`** Details

- Challenge overview
- Leaderboard
- All submissions

4.3. **LFV** in the Duckietown Autolab

The current version of the lane following real robot challenge is `aido-LFV-real-validation`.

Note that to test the performance of your agent on the real robot yourself, you can follow the instructions to run your agent on your Duckiebot

1) **`aido-LFV-real-validation`** Details

- Challenge overview
- Leaderboard
- All submissions

Unit B-5
Challenge `LFI`

The third challenge of the *AI Driving Olympics* is “lane following with intersections” (`LFI`). This challenge is an extension of Challenge `LF` to include map configurations that are not just loops but now contain intersections which must be traversed.

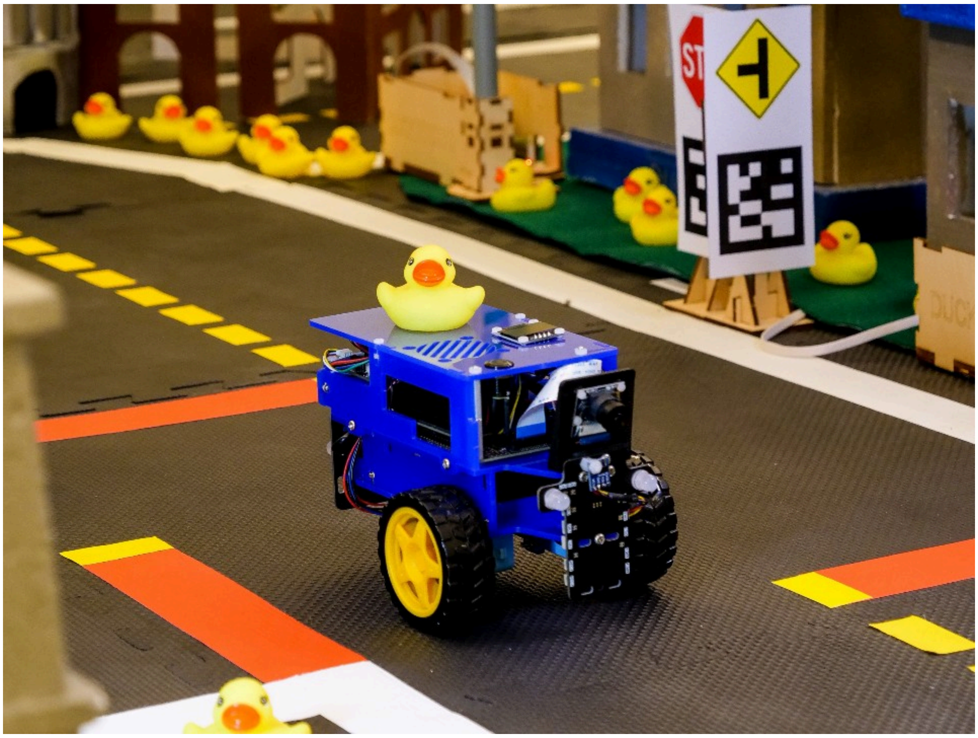


Figure 5.1. A Duckiebot following a lane in a Duckietown with intersections.

Again we ask participants to submit code allowing the Duckiebot to drive on the right-hand side of the street within Duckietown, but now it must also successfully navigate intersections. Due to interactions with other Duckiebots, a successful solution almost certainly not be completely *reactive*.

5.1. `LFI` in Simulation

The current versions of the lane following with vehicles in simulation are `aido-LFI-sim-testing` and `aido2-LF-sim-validation`. These two challenges are identical except for the output that you are allowed to see. In the case of `testing` you will be able to see performance of your agent (Figure 5.2) and you will be able to download the logs and artifacts.

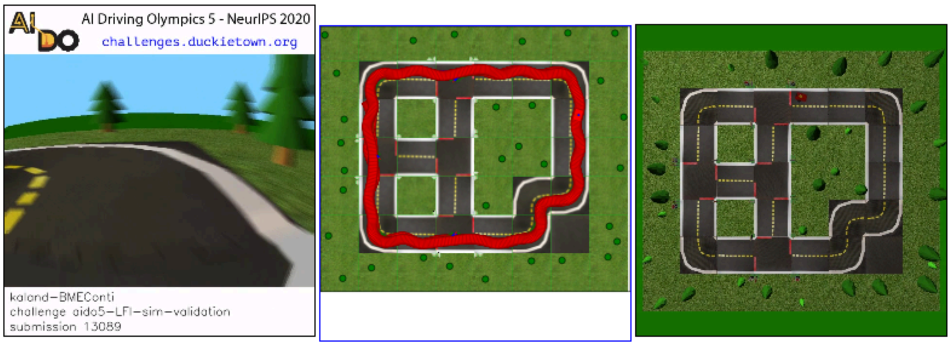

AI Driving Olympics 5 - NeurIPS 2020
challenges.duckietown.org
kalond-BMEConti
challenge aido5-LFI-sim-validation
submission 13089

Figure 5.2. Visual output for a LFI submission.

5.2. Templates and Baselines

To get started, try one of the existing templates, which are minimal setups that do random things but are functions, or the baselines which are instantiations of the templates that implement some algorithms, but probably not in an optimal way. Many of the past AI-DO winners are in the baseline solutions.

1) `aido-LFI-sim-testing` Details

- Challenge overview
- Leaderboard
- All submissions

The details for "experiment manager", "simulator", and "scenario maker" parameters may be of interest and are available here (Under "Details").

2) `aido-LFI-sim-validation` Details

- Challenge overview
- Leaderboard
- All submissions

5.3. `LFI` in the Duckietown Autolab

The current version of the lane following real robot challenge is `aido-LFI-real-validation`.

Note that to test the performance of your agent on the real robot yourself, you can follow the instructions to run your agent on your Duckiebot

1) `aido-LFI-real-validation` Details

- Challenge overview
- Leaderboard
- All submissions

UNIT B-6
Challenge `LFVI-multi-full`

The fourth challenge of the *AI Driving Olympics* is “lane following with dynamic vehicles and intersections” (`LFVI`). This challenge is an extension of Challenge `LF` to include map configurations that are not just loops but now contain intersections which must be negotiated. Your agent will control all the Duckiebots in the map. We make things somewhat simpler by providing directly the state information of the Duckiebots. As a result, this challenge will only be evaluated in simulation

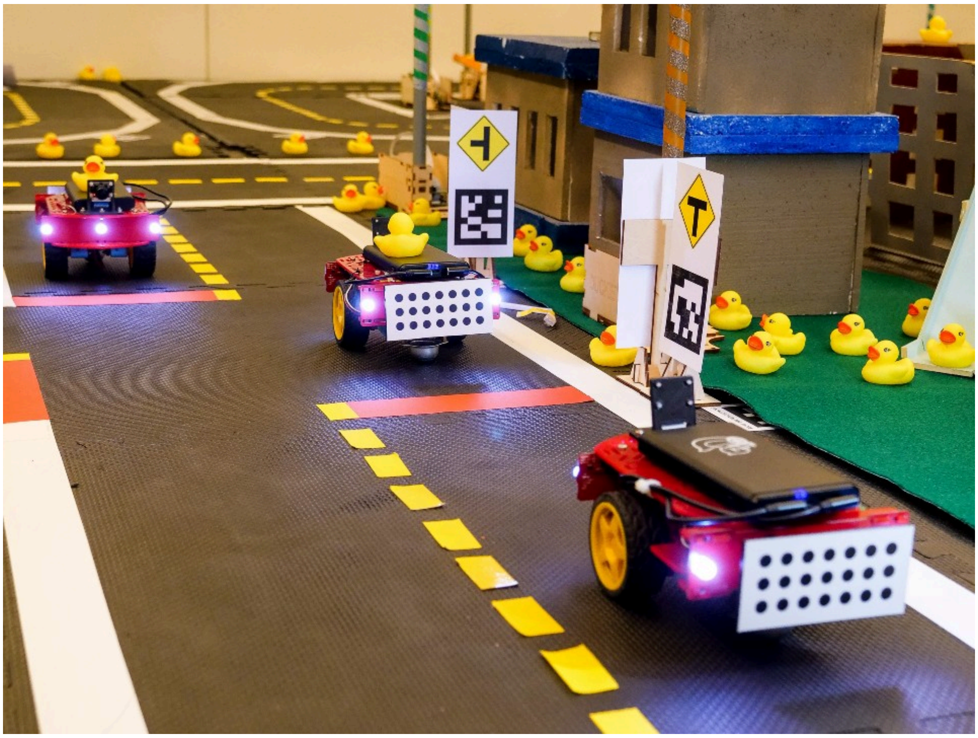


Figure 6.1. A Duckiebot following a lane following in the presence of other vehicles, in a Duckietown with intersections.

Again we ask participants to submit code allowing the Duckiebot to drive on the right-hand side of the street within Duckietown, but now it must also successfully navigate intersections. Due to interactions with other Duckiebots, a successful solution almost certainly not be completely *reactive*.

6.1. `LFVI_multi_full` in Simulation

The current versions of the lane following with vehicles in simulation are `aido-LFVI_multi-sim-testing` and `aido-LFVI_multi-sim-validation`. These two challenges are identical except for the output that you are allowed to see. In the case of `testing` you will be able to see performance of your agent (Figure 6.2) and you will be able to download the logs and artifacts.

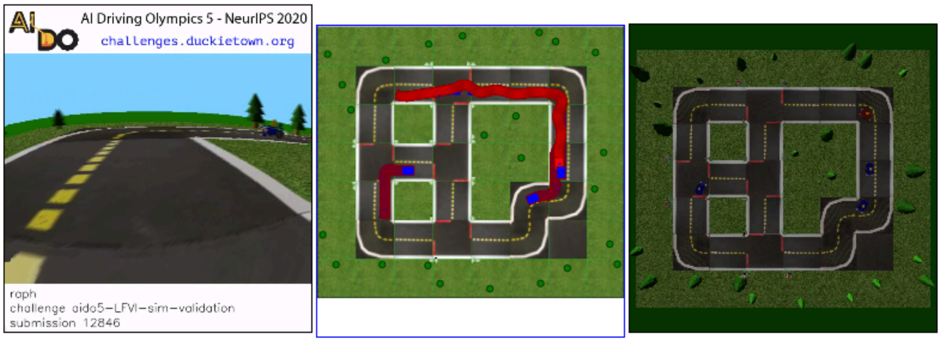

AI Driving Olympics 5 - NeurIPS 2020
challenges.duckietown.org
roph
challenge aido5-LFVI-sim-validation
submission 12846

Figure 6.2. Visual output for a LFVI submission.

6.2. Templates

6.3. Templates and Baselines

To get started, try one of the existing templates, which are minimal setups that do random things but are functions, or the baselines which are instantiations of the templates that implement some algorithms, but probably not in an optimal way. Many of the past AI-DO winners are in the baseline solutions.

Note that in the case of this challenge you will need to update the protocol that is used.

You may also look at the minimal agent with full state information for an example of how to do this.

1) `aido-LFVI_multi-sim-testing` Details

- Challenge overview
- Leaderboard
- All submissions

The details for "experiment manager", "simulator", and "scenario maker" parameters may be of interest and are available here (Under "Details").

2) `aido-LFVI_multi-sim-validation` Details

- Challenge overview
- Leaderboard
- All submissions

PART C

Getting Started

This part describes the necessary steps to get started competing in the AI-DO. It should take about 5-20 minutes depending on your specific setup. In short, the steps are the following:

- Get the needed accounts;
- Make sure you meet the software requirements;
- Make a test submission.

The AI Driving Olympics

@ NeurIPS 2020

aido.duckietown.org

Brought to you by a huge team & many supportive sponsors:

Presenters: Andrea Censi (ETH Zürich), Liam Paull (U. Montréal).

Figure 0.3. Getting Started

At this point you have a fully functioning setup, and you can start to build a solution to the specific challenge that you interested in. In this section, we provide two additional quickstart guides as entry points:

Contents

Unit C-1 Accounts needed

This section describes the accounts that you need before competing.

1.1. Docker Hub account

A Docker Hub account is necessary to submit container images.

Create an account here. Take note of your `USERNAME`.

1.2. Duckietown account

A Duckietown account is necessary to interact with the challenges server.

Create an account here.

1.3. Stack Overflow account

We have a Stack Overflow for Duckietown. We will send you an invitation when you register. Otherwise, please ask us on Slack in the #help-accounts channel.

Unit C-2
Software requirements

This section describes the required software to participate in the competition.

2.1. Supported Operating Systems

1) Ubuntu 20.04

Ubuntu 20.04 is the best supported environment. Earlier version might work. Note that we require an environment with Python 3.8 or higher.

2) Other GNU/Linux versions

Any other GNU/Linux OS with Python of at least version 3.8 should work. However, to streamline assistance, we only support officially Ubuntu.

3) Mac OS X

OS X is well-supported; however we don't have full instructions for certain steps. (There is so much divergence in how OS X environments are configured.)

We suggest to use `pyenv` to install Python 3.8.

4) Windows

Windows is currently not supported. We are working on it! Please let us know on Slack if you can help, in the #devel-wsl channel.

2.2. Docker

Install Docker from these instructions.

If you want to use a GPU for evaluating your submission, edit your `/etc/docker/daemon.json` to include the following options.

```
{
    "default-runtime": "nvidia",

    "runtimes": {
        "nvidia": {
            "path": "nvidia-container-runtime",
            "runtimeArgs": []
        }
    },

    "node-generic-resources": [ "NVIDIA-GPU=0" ]
}
```

> Note: Don't forget that after you install Docker you need to add user to "docker" group:

```
$ sudo adduser `whoami` docker
```

> Note: you likely know about the first two options `default-runtime` and `runtimes`. Be sure to include also the "unusual" option `node-generic-resources`: this is needed because the evaluation uses Docker Compose.

2.3. Git

We need Git and Git LFS.

On Ubuntu you can install both using

```
$ apt-get install git git-lfs
```

2.4. Duckietown Shell

Install the Duckietown Shell using:

```
$ pip3 install --user -U duckietown-shell
```

If you encounter problems look at the Duckietown Shell instructions in the README.

Make sure it is installed by using:

```
$ dts version
```

Set the `daffy` command branch:

```
$ dts --set-version daffy
```

Update the commands using:

```
$ dts update
```

1) Authentication token

Set the Duckietown authentication token using this command:

```
$ dts tok set
```

2) Docker Hub information

Set your Docker Hub username and password using:

```
read -p "docker username: " docker_username
read -p "docker password: " docker_password
dts challenges config --docker-username $docker_username --docker-pass-
word $docker_password
```

You can use an access token instead of a password.

Login to Docker Hub:

```
$ docker login
```

Note: Since November 2, 2020 Docker Hub has implemented tight rate limits for anonymous accounts. If you experience timeouts in Docker or similar problems, it is likely because you have not logged in recently. Note that `docker login` needs to be repeated every 12 hours.

3) Check `dts` configuration

This command checks that you have a good authentication token:

```
$ dts challenges info
```

You should expect an output like:

```
~        You are succesfully authenticated:
~
~                     ID:  your numeric ID
~                   name:  your name
~                  login:  your account name on Duckietown
~                profile:  your website
~
~            You can find the list of your submissions at the page:
~
~                https://challenges.duckietown.org/v4/humans/users/1639
```

UNIT C-3
Make your first submission

This section describes the steps to make your first submission.

KNOWLEDGE AND ACTIVITY GRAPH

Requires: You have set up your accounts.

Requires: You have the software requirement.

Results: You have made a submission to the Lane Following AI-DO challenge, and you know how to try to make it better.

3.1. Checkout the submission repo

Check out the competition template `challenge-aido_LF-template-random`:

```
$ git clone https://github.com/duckietown/challenge-aido_LF-template-
random
```

3.2. Submit

Jump into the directory:

```
$ cd challenge-aido_LF-template-random
```

Submit using:

```
$ dts challenges submit --challenge aido-hello-sim-validation
```

This does the following:

1. Build a Docker container.
2. Push the Docker container.
3. Make contact with the challenge server to send your submission.

The expected output is something along the lines of:

```
Sending build context to Docker daemon  5.632kB
...
...
Successfully created submission SUBMISSION_NUMBER

You can track the progress at: https://challenges.duckietown.org/v4/hu-
mans/submissions/ SUBMISSION_NUMBER

You can also use the command:

   dts challenges follow --submission SUBMISSION_NUMBER
```

where *SUBMISSION_NUMBER* is your submission id.

To understand more about the details of what's happening here see Unit D-1 - Minimal pure-Python Template.

3.3. Monitor the submission

There are 2 ways to monitor the submission:

The first way is to use the web interface, at the URL indicated in the terminal.

The second way is to use the `dts challenges follow` command:

```
$ dts challenges follow --submission SUBMISSION_NUMBER
```

3.4. Look at the leaderboard

The leaderboard for this challenge is available at the URL

`https://challenges.duckietown.org/v4/humans/challenges/aido-hello-sim-validation/leaderboard`

In general all the challenge leaderboards can be viewed at the front page the challenges website.

All available challenges can be viewed in the comprehensive challenges page.

3.5. Local evaluation

You can also evaluate the submission *locally*. This is useful for debugging and development.

Use this command:

```
$ dts challenges evaluate  --challenge aido-hello-sim-validation
```

3.6. Troubleshooting

If any of the commands above don't work, it is likely that something related to Docker permissions is to blame.

If you are using Docker Desktop for Mac OS X you might need to try the following:

> Symptom: `dts challenges submit` fails with a permission error on Mac OS X using Docker Desktop.

Resolution: Disable `gRPC FUSE` in Docker Desktop by going to "Preferences" and unchecking the option "Use gRPC Fuse for file sharing". Select "Apply and Restart" to save the changes.

For other issues please ask us on Slack in the #help-accounts channel.

UNIT C-4
Next steps towards winning the AI-DO

Now that you have made your first submission using the minimal template, you can now move on to the next steps.

Contents

4.1. Understand how the minimal template works

The anatomy of the minimal template is explained in Unit D-1 - Minimal pure-Python Template.

You will understand how the Docker infrastructure works and how to create valid submissions.

4.2. Select the template that you need

The minimal template you tried is a pure-Python template. We offer a few more templates to try if you want to use a framework.

In particular, you could try:

- the TensorFlow template;
- the PyTorch template;
- the ROS template.

4.3. Try the baselines

In Part E - Baseline Algorithms we discuss our “baselines”: submissions that do something smart.

4.4. Understand the rules

You might want to read Part B - The Challenges, which describes in detail how your score is generated for the specific challenges.

4.5. Try one of the harder challenges

In addition to the simple LF challenge you can try the the LFV challenge the LFI challenge or the LFVI-multi challenge where you have access to the state information.

Unit C-5
Run an agent on your Duckiebot

In this page we will describe how to run your submission on your Duckiebot.

Knowledge and activity graph

Requires: You have a Duckiebot. See here for how to acquire a Duckiebot.

Requires: You have built your DB19 (unknown ref opmanual_duckiebot/assembling-duckiebot-db19)

previous **warning** next (8 of 18) index

warning

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

 > I do not know what is indicated by the link '#opmanual_duckiebot/assembling-duckiebot-db19'.
```

Location not known more precisely.

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

or (recommended) DB21 (unknown ref opmanual_duckiebot/assembling-duckiebot-db21)

previous **warning** next (9 of 18) index

warning

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

 > I do not know what is indicated by the link '#opmanual_duckiebot/assembling-duckiebot-db21'.
```

Location not known more precisely.

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

Duckiebot. Evaluations will be performed using `DB21` Duckiebots.

Requires: You have built your Duckietown according to the appearance specification (unknown ref opmanual_duckietown/dt-ops-appearance-specifications)

previous **warning** next (10 of 18) index

warning

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

 > I do not know what is indicated by the link '#opmanual_duckietown/dt-ops-appearance-specifications'.
```

Location not known more precisely.

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

.

Requires: You can connect to your robot wirelessly (unknown ref opmanual_duckiebot/

duckiebot-network)

previous **warning** next (11 of 18) index
warning

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

 > I do not know what is indicated by the link '#op-
manual_duckiebot/duckiebot-network'.
```

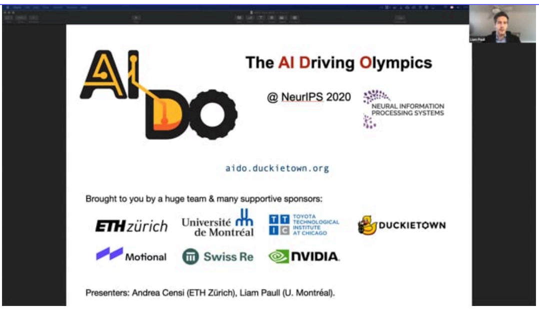
Location not known more precisely.

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

.

Requires: You have made a valid AI-DO submission.

Results: You have run a submission on your physical Duckiebot.



AI DO

The AI Driving Olympics

@ NeurIPS 2020

aido.duckietown.org

Brought to you by a huge team & many supportive sponsors:

Presenters: Andrea Censi (ETH Zürich), Liam Paull (U. Montréal).

Figure 5.1. Running your agent on your Duckiebot tutorial.

Warning: Running your AI-DO submission on your robot is currently only supported on Ubuntu (not Mac OSX).

Warning: If everything's setup right, the procedure is very straightforward. But things can be hard to troubleshoot because they involve networking.

There are two basic modes that you can use to run a submission.

1. From a local submission folder
2. From an existing image (for example one that you submitted to the AI-DO)

5.1. Verifying that your Duckiebot is operational

When you boot your robot it starts to produce camera imagery and wheel encoder data (if it's moving) and waits for incoming motor commands. To verify that your Duckiebot is fully operational, you should follow (unknown ref opmanual_duckiebot/rc-control)

previous **warning** next (12 of 18) index
warning

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

 > I do not know what is indicated by the link '#opman-
ual_duckiebot/rc-control'.
```

Location not known more precisely.

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

and (unknown ref opmanual_duckiebot/read-camera-data)

previous **warning** next (13 of 18) index

warning

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

 > I do not know what is indicated by the link '#opman-
ual_duckiebot/read-camera-data'.
```

Location not known more precisely.

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

.

You should also ensure that your Duckiebot is well calibrated, both camera (unknown ref opmanual_duckiebot/camera-calib)

previous **warning** next (14 of 18) index

warning

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

 > I do not know what is indicated by the link '#opman-
ual_duckiebot/camera-calib'.
```

Location not known more precisely.

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

and wheels (unknown ref opmanual_duckiebot/wheel-calibration)

previous **warning** next (15 of 18) index

warning

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

 > I do not know what is indicated by the link '#opman-
ual_duckiebot/wheel-calibration'.
```

Location not known more precisely.

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

.

5.2. Run a local submission on the Duckiebot

Go into any valid submission folder (i.e., one where you could run `dts submit` and you would make a submission) and run:

```
$ dts duckiebot evaluate --duckiebot_name DUCKIEBOT_NAME
```

5.3. Run an image that is already built on the Duckiebot

```
$ dts duckiebot evaluate --duckiebot_name ![DUCKIEBOT_NAME] --image IMAGE_NAME
```

5.4. Local workflow using the Exercises API

We have also developed a workflow for submitting exercises (unknown ref opmanual_duckiebot/running-exercises)

previous **warning** next (16 of 18) index

warning

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

 > I do not know what is indicated by the link '#opmanual_duckiebot/running-exercises'.
```

Location not known more precisely.

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

in the Duckietown MOOC on EdX that may be useful for your development workflow. Several of the AI-DO templates and baselines are also valid "exercises" and can therefore follow this workflow.

UNIT C-6
Object Detection Dataset

6.1. Download

The dataset can be downloaded from here. We provide annotations and sample scripts for loading the annotations.

6.2. Overview

This dataset consists of 3 categories: traffic cones, duckies, and Duckiebots. All the dataset images were captured with Duckiebot cameras. We use a combination of images from the Duckietown logs database and our own captured logs. Images were captured in different lighting conditions, with different versions of Duckiebot models, and on different Duckietown maps. Below are some statistics and visualizations of our dataset:

Number of images	1956
Number of object categories	3
Number of objects annotated	5068

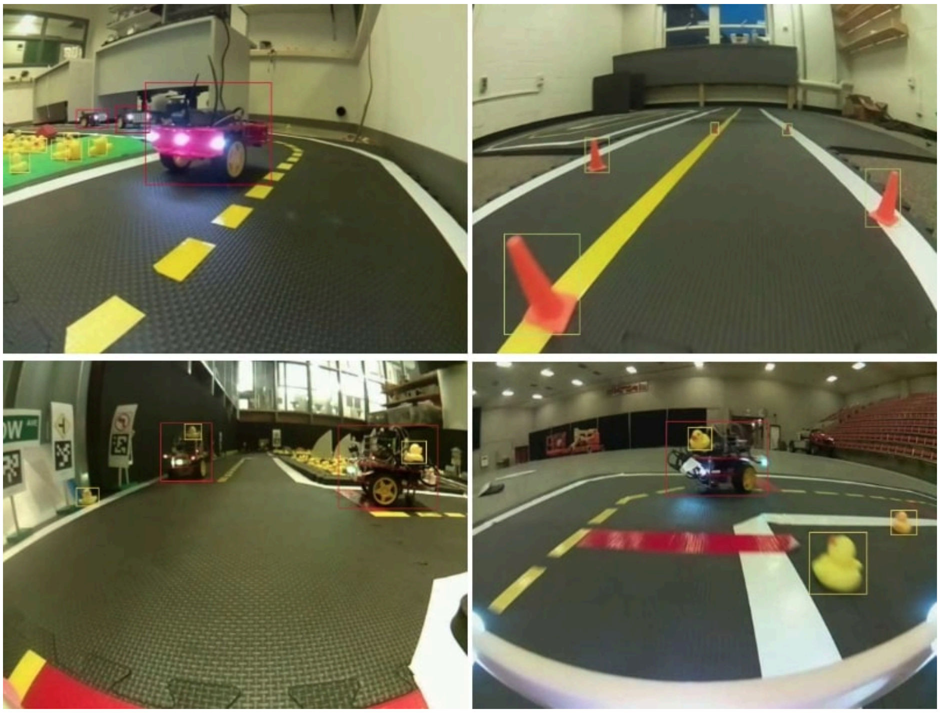


Figure 6.1

6.3. Category Details

1) Traffic Cones

Category name	cone
Number of instances	372
Category id	1

2) Duckies

Category name	duckie
Number of instances	2570
Category id	2

3) Duckiebots

Category name	Duckiebot
Number of instances	2126
Category id	3
Number of old Duckiebot instances	1419
Number of new Duckiebot instances	707

6.4. Data Loading Scripts

We provide some sample scripts for loading in the dataset here.

6.5. Data Collection Procedure

In this work, we first identify the most prominent objects that we see on the roads of Duckietown: duckies, Duckiebots and traffic cones. To begin our data collection procedure, we first identify all useful logs from the Duckietown logs database which contain the objects of interest. We then download and trim these logs so that the videos consist only of frames containing our objects of interest. Finally, we convert our videos to images (frames) while skipping some number of frames between each image to ensure that we get a diverse set of images.

In these logs, there are videos of older versions of Duckiebots with lots of wirings on them (`DB17`). However, new Duckiebots are much cleaner with only the battery visible. To ensure robust detections, we needed to capture this intra-class variation. Thus, we collected our own logs containing the new Duckiebots. In the final dataset, we have merged old and new Duckiebots to ensure that we can detect both variations.

Figure 6.2

6.6. Data Annotation Procedure

We used OpenCV's free CVAT tool to annotate the dataset.

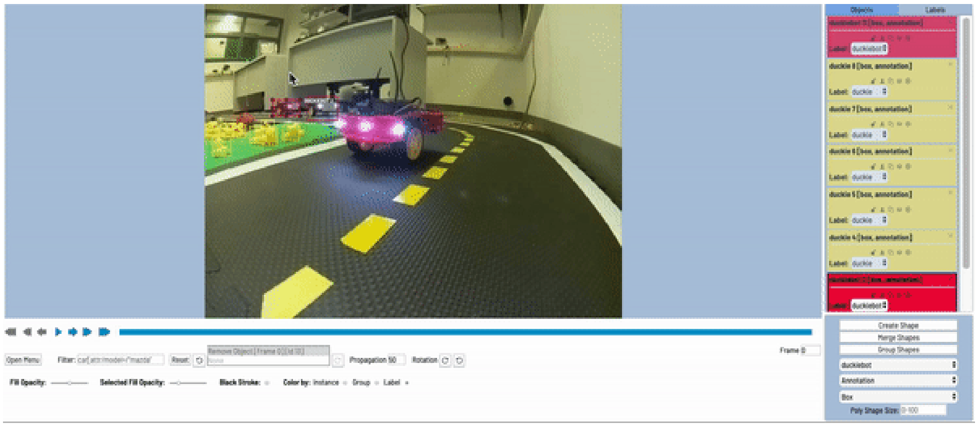


Figure 6.3

Part D
Template Solutions

We provide a set of templates for solutions. These templates are fully functional solutions that don't do anything "smart". They will get you a valid score on the leaderboard, but it's unlikely that it will be very good.

Specifically, we provide the following templates:

- Minimal agent template is the most minimal feasible solution for `LF*` challenges,
- TensorFlow template for making a submission with a tensorflow model to the `LF*` challenges,
- PyTorch template for making a submission with a Pytorch model to the `LF*` challenges,
- ROS template for making a submission using the robot operating system to the `LF*` challenges,

Contents

UNIT D-1
Minimal pure-Python Template

This section describes the contents of the simplest template: a “random” agent. It can be used as a starting point for any of the `LF`, `LFV`, and `LFI` challenges.

KNOWLEDGE AND ACTIVITY GRAPH

Requires: That you have setup your accounts.

Requires: That you meet the software requirement.

Results: You make a submission to all of the `LF*` challenges and can view their status and output.

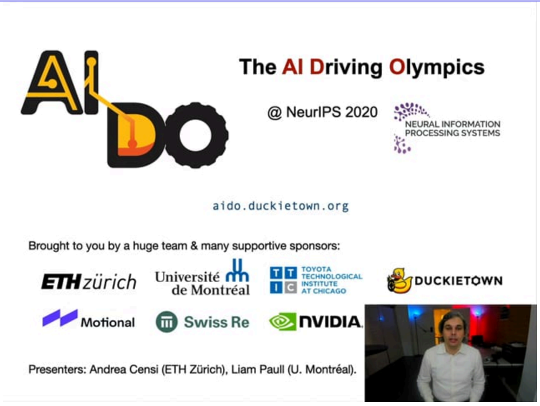

AI DO
The AI Driving Olympics
@ NeurIPS 2020
NEURAL INFORMATION PROCESSING SYSTEMS
aido.duckietown.org
Brought to you by a huge team & many supportive sponsors:
ETH zürich
Université de Montréal
TOYOTA TECHNOLOGICAL INSTITUTE AT CHICAGO
DUCKIETOWN
Motional
Swiss Re
NVIDIA
Presenters: Andrea Censi (ETH Zürich), Liam Paull (U. Montréal).

Figure 1.1. Minimal Template

1.1. Quickstart

Check out the repository:

```
$ git clone git@github.com:duckietown/challenge-aido_LF-template-ran-
dom.git
```

Change into the directory:

```
$ cd challenge-aido_LF-template-random
```

Either make a submission with:

```
$ dts challenges submit --challenge CHALLENGE_NAME
```

where you can find a list of the open challenges here.

Or, run local evaluation with:

```
$ dts challenges evaluate --challenge CHALLENGE_NAME
```

1) Verify your submission(s)

This will make a number of submissions (as described below). You can track the status of these submissions in the command line with:

```
$ dts challenges follow --submission SUBMISSION_NUMBER
```

or through your browser by navigating the webpage: `https://challenges.duckietown.org/v4/humans/submissions/` *SUBMISSION_NUMBER* where *SUBMISSION_NUMBER* should be replaced with the number of the submission which is reported in the terminal output.

1.2. Anatomy of the submission

The submission consists of the following files:

```
submission.yaml
Dockerfile
Makefile
requirements.txt
solution.py
```

1) `submission.yaml`

The file `submission.yaml` contains the configuration for this submission:

```
challenge: [c1,c2]
protocol: aido2_db18_agent-z2
user-label: random_agent
user-payload:
```

- With `challenge` you can list the challenges that you want your submission to be run on.
- The `user-label` can be changed to your liking
- The `protocol` and `user-payload` should probably be left as they are.

2) `requirements.txt`

This file contains any python requirements that are needed by your code.

3) `solution.py`

The `solution.py` solution file illustrates the protocol interface.

The important parts are:

```
def on_received_observations(self, context: Context, data: DB20Observa-
tionsWithTimestamp):
        profiler = context.get_profiler()
        camera: JPGImageWithTimestamp = data.camera
        odometry: DB20OdometryWithTimestamp = data.odometry
        context.info( "camera timestamp: {camera.timestamp}")
        context.info( "odometry timestamp: {odometry.timestamp}")
        with profiler.prof("jpg2rgb"):
            _rgb = jpg2rgb(camera.jpg_data)
```

which reads an image whenever one becomes available, and

```
def on_received_get_commands(self, context: Context, data: GetCommands):
        self.n += 1

        # behavior = 0 # random trajectory
        behavior = 1  # primary motions

        if behavior == 0:
            pwm_left = np.random.uniform(0.5, 1.0)
            pwm_right = np.random.uniform(0.5, 1.0)
            col = RGB(0.0, 1.0, 1.0)
        elif behavior == 1:
            t = data.at_time
            d = 1.0

            phases = [
                (+1, -1, RGB(1.0, 0.0, 0.0)),
                (-1, +1, RGB(0.0, 1.0, 0.0)),
                (+1, +1, RGB(0.0, 0.0, 1.0)),
                (-1, -1, RGB(1.0, 1.0, 0.0)),
            ]
            phase = int(t / d) % len(phases)
            pwm_right, pwm_left, col = phases[phase]

        else:
            raise ValueError(behavior)

        led_commands = LEDSCommands(col, col, col, col, col)
        pwm_commands = PWMCommands(motor_left=pwm_left, mo-
tor_right=pwm_right)
        commands = DB20Commands(pwm_commands, led_commands)
        context.write("commands", commands)
```

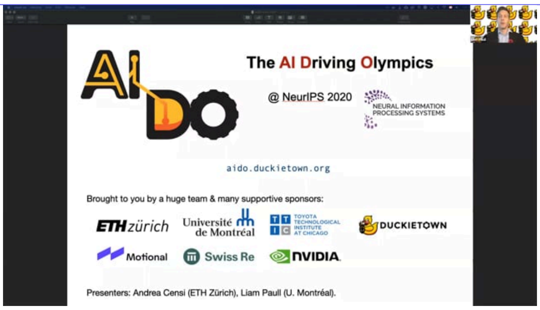
which asks for wheel commands to be sent to the robot. Your code must finish by sending the commands to the robot with the `context.write` command.

UNIT D-2
ROS Template

This section describes the basic procedure for making a submission with an agent using the Robot Operating System. It can be used as a starting point for any of the `LF`, `LFV`, and `LFI` challenges.

KNOWLEDGE AND ACTIVITY GRAPH

Requires: That you have setup your accounts.
Requires: That you meet the software requirement.
Requires: That you have a basic understanding of ROS.
Results: You make a submission to all of the `LF*` challenges and can view their status and output.



The AI Driving Olympics

@ NeurIPS 2020

aido.duckietown.org

Brought to you by a huge team & many supportive sponsors:

Presenters: Andrea Censi (ETH Zürich), Liam Paull (U. Montréal).

Figure 2.1. ROS template

2.1. Quickstart

Clone the template repo:

```
$ git clone git@github.com:duckietown/challenge-aido_LF-template-ros.git
```

Change into the directory:

```
$ cd challenge-aido-LF-template-ros
```

Either make a submission with:

```
$ dts challenges submit --challenge CHALLENGE_NAME
```

where you can find a list of the open challenges here.

Or, run local evaluation with:

```
$ dts challenges evaluate --challenge CHALLENGE_NAME
```

1) Verify the submission:

This will make a number of submissions (as described below). You can track the status of these submissions in the command line with:

```
$ dts challenges follow --submission SUBMISSION_NUMBER
```

or through your browser by navigating the webpage: `https://challenges.duckietown.org/v4/humans/submissions/` *SUBMISSION_NUMBER*

where *SUBMISSION_NUMBER* should be replaced with the number of the submission which is reported in the terminal output.

2.2. Anatomy of the submission

The submission consists of all of the basic files that required for a basic submission. Below we will highlight the specifics with respect to this template.

There are also a few other **new** files and folders in this submission:

```
launchers/
submission_ws/
```

and additionally the `solution.py` is inside the `submission_ws` folder and `Dockerfile` have changed. We will describe each of these in detail.

> **Note:** If you don't care about the details, or just want to get started, you can start by adding new ROS packages into the `submission_ws`.

1) Dockerfile

The main update here is that we build your catkin workspace inside (the `submission_ws` folder) in the Dockerfile:

```
RUN . /opt/ros/${ROS_DISTRO}/setup.sh && \
    . ${CATKIN_WS_DIR}/devel/setup.bash  && \
    catkin build --workspace /code/submission_ws
```

Also note that instead of just running `solution.py` when we enter the container, we now run a "launcher" (in the `launchers` folder) called `run_and_start.sh`. For details see Subsection 2.2.4 - `launchers/`.

Also note that in this Dockerfile we are not copying the entire directory over, instead we are copying files individually (this is actually more efficient). So if you add new files that you are using that are outside of the `submission_ws` and `launchers` folders, you will have to add additional `COPY` commands.

2) `solution.py`

You probably don't need to change this file.

We instantiate a `ROSAgent()` (see Subsection 2.2.3 - `rosagent.py`) and this becomes the object that handles interfacing with the ROS interface. This includes the publishing of imagery and encoder data to ROS:

```
def on_received_observations(self, data: DB20ObservationsWithTimestamp,
context: Context):
        camera = data.camera
        odometry = data.odometry
        # context.info(f'received obs camera {camera.timestamp} odome-
try {odometry.timestamp}')

        if camera.timestamp != self.last_camera_timestamp or True:
            self.agent.publish_img(camera.jpg_data, camera.timestamp)
            self.agent.publish_info(camera.timestamp)
            self.last_camera_timestamp = camera.timestamp

        if odometry.timestamp != self.last_odometry_timestamp or True:
            self.agent.publish_odometry(
                odometry.resolution_rad, odometry.axis_left_rad, odome-
try.axis_right_rad, odometry.timestamp
            )
            self.last_odometry_timestamp = odometry.timestamp
```

Notice now that the protocol includes timestamps which are used to tag the data, and that a new camera image is not published if the timestamp does not change.

3) `rosagent.py`

You probably don't need to change this file.

`rosagent.py` sets up a class that can be used to interface with the rest of the ROS stack. It is for all intents and purposes a fully functional ROS node except that it isn't launched through ROS, it is instantiated in code. This class takes care of a few useful things, such as getting the correct camera calibration files, subscribing to control commands and sending them to your robot (real or simulated), as well as retreiving the sensor data from the robot and publishing it to ROS.

The main functions are:

- `def publish_img(self, obs: bytes, timestamp: float):`, which takes the camera observation from the environment, and publishes it to the topic that you specify in the constructor of the `ROSAgent`
- `def publish_odometry(self, resolution_rad: float, left_rad: float, right_rad: float, timestamp: float):`, which take the encoder data from the robot, and publishes it to the topic specified in the constructor of the `ROSAgent`.
- `def _ik_action_cb(self, msg):`, listens on the inverse kinematics action topic, and assigns it to `self.action`.

4) `launchers/`

The bash scripts in the `launchers` directory are there to help you get everything started when you run your container. In this template there is only `run_and_start.sh`:

```
#!/bin/bash

source /environment.sh

source /opt/ros/noetic/setup.bash
source /code/catkin_ws/devel/setup.bash --extend
source /code/submission_ws/devel/setup.bash --extend

set -eux

dt-exec-BG roscore

dt-exec-BG roslaunch --wait random_action random_action_node.launch
dt-exec-FG roslaunch --wait agent agent_node.launch || true

copy-ros-logs
```

You are free to modify this as you see fit, but a few things are important to consider.

1. The order that we `source` things matters. If we have a package with the same name in two workspaces, ROS will run whichever one got sourced last.

2. If you don't put things in the background (with `dt-exec-BG`) then if those commands don't end, subsequent commands will not get run.

3. The `--wait` flag in the `roslaunch` command is recommended so that `roslaunch` will wait until the `roscore` has finished initializing.

5) `submission_ws/`

This is a standard ROS catkin workspace. You can populate it with ROS packages. You will notice that the `random_action` package is already in the workspace. This can be used as a template for creating more packages. The main elements are launch files in the `launch` folder (you will see the `random_action_node.launch` which is launched by the `run_and_start.sh` launcher), the `src` folder which contains the ROS nodes, and the `include` folder which contains your python includes (you can also write nodes in C++ or other languages if you prefer).

UNIT D-3
TensorFlow Template

This section describes the basic procedure for making a submission with a model trained in using TensorFlow. It can be used as a starting point for any of the `LF`, `LFV`, and `LFI` challenges.

KNOWLEDGE AND ACTIVITY GRAPH

Requires: That you have setup your accounts.

Requires: That you meet the software requirement.

Requires: 9 GB free space.

Results: You make a submission to all of the `LF*` challenges and can view their status and output.

The AI Driving Olympics
@ NeurIPS 2020
NEURAL INFORMATION PROCESSING SYSTEMS
aido.duckietown.org
& many supportive sponsors:
TOYOTA TECHNOLOGICAL INSTITUTE AT CHICAGO
DUCKIETOWN
Motional
Swiss Re
NVIDIA
Today's Presenters: Chude Qian (U. Toronto), Moustafa ElaRaby (U. Montréal)

Figure 3.1. TensorFlow Template

3.1. Quickstart

Clone the template repo:

```
$ git clone git@github.com:duckietown/challenge-aido_LF-template-tensor-
flow.git
```

Change into the directory:

```
$ cd challenge-aido_LF-template-tensorflow
```

Either make a submission with:

```
$ dts challenges submit --challenge CHALLENGE_NAME
```

where you can find a list of the open challenges here.

Or, run local evaluation with:

```
$ dts challenges evaluate --challenge CHALLENGE_NAME
```

1) Verify your submission(s)

This will make a number of submissions (as described below). You can track the status of these submissions in the command line with:

```
$ dts challenges follow --submission SUBMISSION_NUMBER
```

or through your browser by navigating the webpage: `https://challenges.duckietown.org/v4/humans/submissions/` *SUBMISSION_NUMBER*
where *SUBMISSION_NUMBER* should be replaced with the number of the submission which is reported in the terminal output.

3.2. Anatomy of the submission

The submission consists of all of the basic files that required for a basic submission. Below we will highlight the specifics with respect to this template.

1) `solution.py`

The only difference in `solution.py` is that we are initializing our model:

```
from model import TfInference
    # define observation and output shapes
    self.model = TfInference(observation_shape=(1,) + expect_shape,
                                 # this is the shape of the image we
get.
                                 action_shape=(1, 2),  # we need to out-
put v, omega.
                                 graph_location='tf_models/')  # this
is the folder where our models are stored.
    self.current_image = np.zeros(expect_shape)
```

and then we call our model to compute an action with the following code:

```
def compute_action(self, observation):
        action = self.model.predict(observation)
        return action.astype(float)
```

Note that we also can require the presence of a GPU with the environment variable `AIDO_REQUIRE_GPU` and then the solution will fail if a GPU is not found.

2) Model files

The other additional files are the following:

```
tf_models/
model.py
```

The directory `tf_models/` contains the Tensorflow learned models (the ones that you have trained).

The `model.py` code is the code that runs the Tensorflow model.

UNIT D-4

PyTorch Template

This section describes the basic procedure for making a submission with a model trained in using PyTorch.

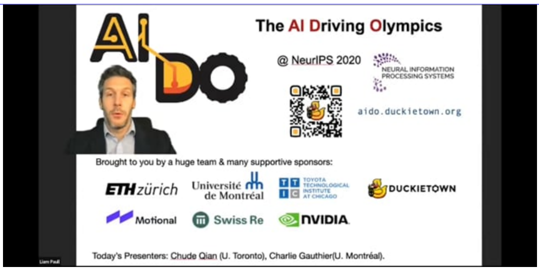
It can be used as a starting point for any of the `LF`, `LFV_multi`, and `LFI` challenges.

KNOWLEDGE AND ACTIVITY GRAPH

Requires: That you have setup your accounts.

Requires: That you meet the software requirement.

Results: You make a submission to all of the `LF*` challenges and can view their status and output.



The AI Driving Olympics

@ NeurIPS 2020

aido.duckietown.org

Brought to you by a huge team & many supportive sponsors:

Today's Presenters: Chude Qian (U. Toronto), Charlie Gauthier(U. Montréal).

Figure 4.1. PyTorch Template

4.1. Quickstart

Clone the template repo:

```
$ git clone git://github.com/duckietown/challenge-aido_LF-template-py-
torch.git
```

Change into the directory:

```
$ cd challenge-aido_LF-template-pytorch
```

Run the submission:

Either make a submission with:

```
$ dts challenges submit --challenge CHALLENGE_NAME
```

where you can find a list of the open challenges here.

Or, run local evaluation with:

```
$ dts challenges evaluate --challenge CHALLENGE_NAME
```

1) Verify the submission(s)

This will make a number of submissions (as described below). You can track the status of these submissions in the command line with:

```
$ dts challenges follow --submission SUBMISSION_NUMBER
```

or through your browser by navigating the webpage: `https://challenges.duckietown.org/v4/humans/submissions/` *SUBMISSION_NUMBER*

where *SUBMISSION_NUMBER* should be replaced with the number of the submission which is reported in the terminal output.

4.2. Anatomy of the submission

The submission consists of all of the basic files that required for a basic submission. Below we will highlight the specifics with respect to this template.

1) `solution.py`

The only differences in `solution.py` (the python script that is run by our submission) are:

- We conditionally load the model in the initializaiton procedure:

```
self.model = DDPG(state_dim=self.preprocessor.shape, action_dim=2,
max_action=1, net_type="cnn")
self.current_image = np.zeros((640, 480, 3))

if load_model:
    logger.info('PytorchRLTemplateAgent loading models')
    fp = model_path if model_path else "model"
    self.model.load(fp, "models", for_inference=True)
```

- We abort if no GPU is detected and the environment variable `AIDO_REQUIRE_GPU`.
- We are calling our model to compute an action with the following code:

```
def compute_action(self, observation):
        action = self.model.predict(observation)
        return action.astype(float)
```

4.3. Model files

The other addition files are the following:

```
wrappers.py
model.py
models
```

`wrappers.py` contains a simple wrapper for resizing the input image. `model.py` is used for training the model, and the models are stored in `models`.

Part E
Baseline Algorithms

To help competitors get started, we have implemented some baseline algorithms. These can be built on or used for inspiration. At present, all of these baseline algorithms are for the `LF*` challenges:

Contents

UNIT E-1
Duckietown Baseline

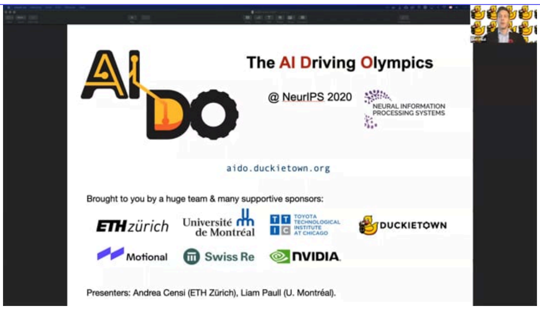
This section describes the basic procedure for making a submission using the Robot Operating System and the Duckietown software stack.

KNOWLEDGE AND ACTIVITY GRAPH

Requires: That you have made a submission with the ROS template and you understand how it works.

Requires: You already know something about ROS.

Results: You have a competitive submission.



The AI Driving Olympics

@ NeurIPS 2020

aido.duckietown.org

Brought to you by a huge team & many supportive sponsors:

Presenters: Andrea Censi (ETH Zürich), Liam Paull (U. Montréal).

Figure 1.1. ROS template

1.1. Quickstart

Clone this repo

```
$ git clone git@github.com:duckietown/challenge-aido_LF-baseline-ducki-
etown.git
```

Change into the directory:

```
$ cd challenge-aido_LF-baseline-duckietown
```

Test the submission, either locally with:

```
$ dts challenges evaluate --challenge CHALLENGE_NAME
```

or make an official submission when you are ready with

```
$ dts challenges submit --challenge CHALLENGE_NAME
```

You can find the list of challenges here. Make sure that it is marked as “Open”.

1.2. Baseline Details

The “Duckietown” baseline is based on the ROS template.

1) Dockerfile

One important fact of the Dockerfile is that we use a “multi-stage build”:

```
FROM ${DOCKER_REGISTRY}/duckietown/dt-car-interface:${BASE_TAG} AS dt-
car-interface

FROM ${DOCKER_REGISTRY}/duckietown/challenge-aido_lf-template-
ros:${BASE_TAG} AS template

FROM ${DOCKER_REGISTRY}/duckietown/dt-core:${BASE_TAG} AS base
```

This allows us to take some elements from each of the first two base images, and copy them into the `dt-core` image:

```
COPY --from=dt-car-interface ${CATKIN_WS_DIR}/src/dt-car-interface
${CATKIN_WS_DIR}/src/dt-car-interface
COPY --from=template /data/config /data/config
COPY --from=template /code/rosagent.py .
```

As a result, we have the calibration files (from `/data/config`) as well as the `rosagent.py` from the `challenge-aido_lf-template-ros` and all the source files from the `dt-car-interface` image.

We also get everything that is in the `dt-core` image.

The remainder of the Dockerfile is very similar to the Dockerfile in the ROS template.

2) `solution.py`

There is no `solution.py` because it is inherited from the ROS template. In the event that you wanted to, for example, change the launcher that was run in the final `CMD` line.

3) `launchers/`

There is only one “launcher”, and it deviates slightly from the one in the ROS template:

```
#!/bin/bash

source /environment.sh

source /opt/ros/noetic/setup.bash
source /code/catkin_ws/devel/setup.bash --extend
source /code/solution/devel/setup.bash --extend
source /code/submission_ws/devel/setup.bash --extend

set -eux

dt-exec-BG roscore

dt-exec-BG roslaunch --wait car_interface all.launch veh:="${VEHI-
CLE_NAME}"
dt-exec-BG roslaunch --wait duckietown_demos lane_following.launch

sleep 5 # for some reason we still need this so that nodes can startup
dt-exec-BG roslaunch --wait duckietown_demos set_state.launch
veh:="${VEHICLE_NAME}" state:="LANE_FOLLOWING"

rostopic list
# foreground
dt-exec-FG roslaunch --wait agent agent_node.launch || true
rostopic list
copy-ros-logs
```

Here we launch the `lane_following.launch` launch file from the `duckietown_demos` package. We don't go into the intricate details of everything that is run in this launch file here, but some of the more consequential nodes which are getting launched are the following:

- line_detector_node: Used to detect the lines in the image.
- ground_projection_node: Used to project the lines onto the ground plane using the camera extrinsic calibration.
- lane_filter_node: Used to take the ground projected line segments and estimate the Duckiebot's position and orientation in the lane.
- lane_controller_node: Used to take the estimate of the robot and generate a reference linear and angular velocities for the Duckiebot.

In the event that you wanted to, for example change the launcher that was run in the final `CMD` line.

4) `submission_ws/`

The `submission_ws` folder contains all the new ROS packages that you would like to include in your submission. It is currently empty, but there is a reference package included in the ROS template.

> **Note:** Importantly, your `submissions_ws` is sourced **after** the existing `catkin_ws` that is included in `dt-core`. As a result, if you include a node and package in your `submission_ws` *with the same name* as one in `dt-core`, the one in `submission_ws` will get executed. This is convenient because it means that, as long as you adhere to the same subscriptions and publications, you don't need to define any new launch file, `lane_following.launch` will automatically launch your newly written node.

1.3. Local Development Workflow

For rapid local development, you can make use of the `dts exercises` API (unknown ref opmanual_duckiebot/running-exercises)

previous **warning** next (17 of 18) index

warning

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

 > I do not know what is indicated by the link '#opman-
ual_duckiebot/running-exercises'.
```

Location not known more precisely.

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

, developed to build and test exercises and assignments in class settings.

1) Building your Code

From inside the `challenge-aido_LF-baseline-duckietown` folder, you can start by building your code with:

```
$ dts exercises build
```

This performs `catkin build` inside a docker container. If you go inside the `submission_ws` folder you will notice that there are more folders that weren't there before. These are build artifacts that persist from the building procedure because of mounting.

2) Running in Simulation

You can run your current solution in the gym simulator with:

```
 $ dts exercises test --sim
```

Then you can look at what's happening by looking through the browser at http://localhost:8087. This will open a noVNC desktop. In it, open up the `rqt_image_view`, resize it, and choose `/agent/camera_node/image/compressed` in the dropdown. You should see the image from the robot in the simulator.

You might want to launch a virtual joystick by opening a terminal and doing:

```
$ dt-launcher-joystick
```

By default the Duckiebot is in joystick control mode, so you can freely drive it around. You can also set it to `LANE FOLLOWING` mode by pushing the `a` button when you have the virtual joystick active. If you do so you will see the robot move forward slowly and never turn.

At the same time, you can see a birds eye overview of the Duckiebot on the track though the browser at http://localhost:8090.

3) Testing Your Algorithm on the Robot

If you are using a Linux laptop, you have two options, local (i.e., on your laptop) and remote (i.e., on the Duckiebot). To run "locally"

```
$ dts exercises test --duckiebot_name ROBOT_NAME --local
```

To run on the Duckiebot:

```
$ dts exercises test --duckiebot_name ROBOT_NAME
```

In both cases you should still be able to look at things through noVNC by pointing your browser to http://localhost:8087 . If you are running on Linux, you can load up the virtual joystick and start lane following as above.

Warning: If you are Mac user unfortunately you should not use the `--local` flag

Starting Lane Following on Mac:

Since we can't publish from Mac and have it be received by ROS, we have to do something slightly different. In a new terminal on your Mac do:

```
$ docker -H ROBOT_NAME.local exec agent launchers/start_lane_follow-
ing.sh
```

This will run the `start_lane_following.sh` bash script inside the agent container which initiates `LANE_FOLLOWING` mode.

Similarly, you can stop your Duckiebot from lane following by doing:

```
$ docker -H ROBOT_NAME.local exec agent launchers/stop_lane_follow-
ing.sh
```

You could also do an equivalent thing through the Portainer interface in the dashboard by opening a new terminal in your agent container and running the corresponding launcher.

4) How to Improve your Submission

A good way to get started could be to copy one of the packages defined in the Duckietown dt-core repo or the Duckietown dt-car-interface repo into the `submission_ws` folder and modify it. Note that your modified package will automatically get run because of the order of the sourcing of the catkin workspaces in the `run_and_start.sh` launch file.

If you would like to add a new package and node that includes a functionality not already run by `lane_following.launch` or you would like to change the connectivity of interfaces of these nodes, then you will also need:

- to write your own launch file that launches your node and also all of the other nodes from the base images that you would still like to use.
- to modify the launch file `run_and_start.sh` so that it launches your newly created launchfile. You could equally define a new launchfile, but then make sure that it gets executed in the last line of your `Dockerfile`.

5) Other Possibly Useful Utilities

All of the normal ROS debugging utilities are available to you through the noVNC desktop. For example, You might also explore the other outputs that you can look at in `rqt_image_view`.

Also useful are some debugging outputs that are published and visualized in `RViz`. You can open `RViz` through the terminal in the noVNC desktop by typing:

```
$ rviz
```

In the window that opens click "Add" the switch to the topic tab, then find the `segment_markers`, and you should see the projected segments appear. Do the same for the `pose_markers`.

Another tool that may be useful is `rqt_plot` which also can be opened through the terminal in noVNC. This opens a window where you can add "Topics" in the text box at the top left and then you will see the data get plotted live.

All of this data can be viewed as data through the command line also. Take a look at all of the `rostopic` command line utilities.

UNIT E-2
Reinforcement Learning

This section describes the basic procedure for making a submission with a model trained in simulation using reinforcement learning with PyTorch.

KNOWLEDGE AND ACTIVITY GRAPH

Requires: That you have made a submission with the PyTorch template.

Requires: You should install CUDA10.2+ locally. This baseline works with CUDA 11, and it should also work with CUDA 10.2.

Requires: Patience, training RL agents is not easy.

Results: You have a functional agent trained with RL. Your expectations in regards to end-to-end RL's capabilities should be realistic.

Before getting started, you should be aware that RL is very much an active area of research. Simply getting a successful turn with this baseline should be celebrated. It is still provided to you because this implementation is a good stepping point to other algorithms. We also assume here that you are relatively familiar with the basics of reinforcement learning. There are many tutorials and resources, and even complete courses, online for learning about RL, but for a succinct introduction, you can check out the Reinforcement Learning lecture from the IFT6757 class at the University of Montreal, or try our reinforcement learning Jupyter notebook which is in the Duckietown exercises repository.

You should also make sure you have access to good hardware. A recent graphics card (probably GTX1060+) is a must, and more than 8GB of RAM is required.

2.1. Quickstart

Clone this repo

```
$ git clone git@github.com/duckietown/challenge-aido_LF-baseline-sim-py-
torch.git
```

Change into the directory:

```
$ cd challenge-aido_LF-baseline-sim-pytorch
```

Test the submission, either locally with:

```
$ dts challenges evaluate --challenge CHALLENGE_NAME
```

or make an official submission when you are ready with

```
$ dts challenges submit --challenge CHALLENGE_NAME
```

You can find the list of challenges here. Make sure that it is marked as “Open”.

2.2. How to Train your Policy

The previous uses the model that is included in the baseline repository. You are going to want to train your own policy.

To do so:

Change into the directory:

```
$ cd challenge-aido_LF-baseline-RL-sim-pytorch
```

Install this package:

```
$ pip3 install -e .
```

and the `gym-duckietown` package:

```
$ pip3 install -e git://github.com/duckietown/gym-ducki-
etown.git@daffy#egg=gym-duckietown
```

Note: Depending on your configuration, you might need to use pip instead of pip3

Change into the `duckietown_rl` directory and run the training script

```
$ cd duckietown_rl
$ python3 -m scripts.train_cnn.py --seed 123
```

When it finishes, try it out (make sure you pass in the same seed as the one passed to the training script)

```
$ python3 -m scripts.test_cnn.py --seed 123
```

2.3. How to submit the trained policy

Once you’re done training, you need to copy your model and the saved weights of the policy network.

Specifically if you use this repo then you need to copy the following artifacts into the corresponding locations of the root directory:

- `duckietown_rl/ddpg.py` and rename to `model.py`
- `scripts/pytorch_models/DDPG_XXX_actor.pth` and `DDPG_XXX_critic.pth` and rename to `models/model_actor.pth` and `models/model_critic.pth` respectively, where

`XXX` is the seed of your best policy

Also, make sure that the root-level `wrappers.py` contains all the wrappers you used in `duckietown_rl/wrappers.py`.

Then edit the `solution.py` file over to make sure everything is loaded correctly (i.e., all the imports point to the right place).

Finally, you `evaluate` or `submit` your agent using the process described above in the Quickstart.

2.4. How to improve your policy

Here are some ideas for improving your policy:

- Check out the `DtRewardWrapper` and modify the rewards (set them higher or lower and see what happens)
- Try resizing the images. Make them smaller to speed up training, or bigger for ensuring that your RL agent can extract everything it can from them. You will need to also edit the layers in `ddpg.py` accordingly.
- Try making the observation image grayscale instead of color.
- Try stacking multiple images, like 4 monochrome images instead of 1 color image. You will need to also edit the layers in `ddpg.py` accordingly.
- You can also try increasing the contrast in the input to make the difference between road and road-signs clearer. You can do so by adding another observation wrapper.
- Cut off the horizon from the image (and correspondingly change the convnet parameters).
- Check out the default hyperparameters in `duckietown_rl/args.py` and tune them. For example increase the `expl_noise` or increase the `start_timesteps` to get better exploration.
- (more sophisticated) Use a different map in the simulator, or - even better - use randomized maps. But be mindful that some maps include obstacles on the road, which might be counter-productive to a `LF` submission.
- (more advanced) Use a different/bigger convnet for your actor/critic. And add better initialization.
- (very advanced) Use the ground truth from the simulator to construct a better reward.
- (extremely advanced) Use an entirely different training algorithm - like PPO, A2C, or DQN. But this might take significant time, even if you're familiar with the matter.

2.5. Sim2Real Transfer (Optional)

You should try your agent on the real Duckiebot.

It is possible, even likely, that your agent will not generalize well to the real environment. One approach to mitigate this problem is to randomize the simulator environment during training, in the hope that this improves generalization. This approach is referred to as "Domain Randomization".

To implement this, you will need to modify the `env.py` file. You'll notice that we launch the `Simulator` class from `gym-duckietown`. When we take a look at the constructor, you'll notice that we aren't using all of the parameters listed. In particular, the three you should focus on are:

- `map_name`: What map to use; hint, take a look at gym_duckietown/maps for more choices
- `domain_rand`: Applies domain randomization, a popular, black-box, sim2real technique
- `randomized_maps_on_reset`: Slows training time, but increases training variety.

Mixing and matching different values for these will help you improve your training diversity, and thereby improving your evaluation robustness.

If you're interested in more advanced techniques, like learning a representation that is a bit easier for your network to work with, or one that transfers better across the simulation-to-reality gap, there are some alternative, more advanced methods you may be interested in trying out.

2.6. Training headless

Should you want to train on a server, you will notice that the simulator requires an X server to run. Fear not, however, as we can use a fake X server for it.

```
$ xvfb-run -s "-screen 0 1400x900x24" python3 -m scripts.train_cnn.py --
seed 123
```

That way, we trick the simulator into thinking that an X server is running. And, to be honest, from its point of view, it's actually true!

2.7. Controlling which GPU is being used

Your machine might have more than one GPU. To select the nth instead of the 0th, you can use

```
$ CUDA_VISIBLE_DEVICES=n python3 -m scripts.train_cnn.py --seed 123
```

This is, of course, combinable with running on a server

```
$ CUDA_VISIBLE_DEVICES=n xvfb-run -s "-screen 0 1400x900x24" python3 -m
scripts.train_cnn.py --seed 123
```

UNIT E-3
Behavior Cloning

In this part, you can find the required steps to make a submission based on Behavior Cloning with Tensorflow for the lane following task, using data from real world or simulator data. It can be used as a strong starting point for any of the challenges.

KNOWLEDGE AND ACTIVITY GRAPH

Requires: That you have made a submission with the tensorflow template.

Results: You could win the AI-DO!

The AI Driving Olympics
@ NeurIPS 2020
aido.duckietown.org
& many supportive sponsors:
DUCKIETOWN
Motional
Swiss Re
NVIDIA
Today's Presenters: Chude Qian (U. Toronto), Moustafa ElaRaby (U. Montréal)

Figure 3.1. Behavior Cloning

3.1. Introduction

This baseline refers to the approach for behavior cloning for autonomous vehicles described in this paper: End to End Learning for Self-Driving Cars. It is created by Frank (Chude Qian) for his submission to AI-DO 3 at NeurIPS 2019. The submission was very successful on simulator challenge, however, it was not the best for real world challenges.

A detailed description on the specific implementation for this baseline can be found on the summary poster here: Teaching Cars to Drive Themselves.

3.2. Quickstart

Clone the baseline Behavior Cloning repository:

```
$ git clone -b daffy https://github.com/duckietown/challenge-aido_LF-baseline-behavior-cloning.git

$ cd challenge-aido_LF-baseline-behavior-cloning
```

The code is structured into 5 folders:

1. Teach your Duckiebot to drive itself in `duckieSchool`.
2. *(Optional)* Store all the logs that can be used for training using `duckieLog`.
3. Train your model using tensorflow in `duickieTrainer`.

4. *(Optional)* Hold all previous models you generated in `duckieModels`.
5. Submit your submission via `duckieChallenger` folder.

3.3. The duckieSchool

In side this folder you find two types of `duckieSchool`: simulator based `duckieGym` and real robot based `duckieRoad`.

1) Installing duckietown Gym

To install duckietown Gym and all the necessary dependencies:

```
pip3 install  -r requirements.txt
```

2) Use joystick to drive

Before you use the script, make sure you have the joystick connected to your computer.

To run the script, use the following command:

```
$ python3 human.py
```

The system utilizes an Xbox One S joystick to drive around. Left up and down controls the speed and right stick left and right controls the velocity. Right trigger enables the "DRS" mode and allows the vehicle to drive full speed forward. (Note there are no angular acceleration when this mode is enabled).

In addition, every 1500 steps in simulator, the recording will pause and playback. You will have the chance to review the result and decide whether to keep the log or not. The logs are recorded in two formats: `raw_log` saves all the raw information for future reprocessing, and `traning_data` saves the directly feedable log.

3) Options for joystick script

For driving a Duckiebot with a joystick in a simulator, you have the following options:

1. `--env-name`: currently the default is `None`.
2. `--map-name`: This sets the map you choose to run. Currently, it is set as `small_loop_cw`.
3. `--draw-curve`: This draw the lane following curve. Default is set as `False`. However, if you are new to the system, you should familiarize yourself with enabling this option as `True`.
4. `--draw-bbox`: This helps draw out the collision detection bounding boxes. Default is set as `False`.
5. `--domain-rand`: This enables domain randomization. Default is set as `True`.
6. `--playback`: This enables playback after each record section for you to inspect the log you just took. Default is set as `True`.
7. `--distortion`: This enables distortion to let the view as fisheye lens. Default is set as `True`.

8. `--raw_log`: This enables recording also a high resolution version of the log instead of the down-sampled version. Default is set as `True`. **Note: if you disable this option, playback will be disabled too.**

9. `--steps`: This sets how many steps to record once. Default is set as `1500`.

10. `--nb-episodes`: This controls how many episodes (a.k.a. sessions) you drive.

11. `--logfile`: This specifies where you can store your log file. Default will just save the log file in the current folder.

12. `--downscale`: This option currently is disabled.

13. `--filter-bad-data`: This option allows you to only log driving that is better than the last state. It uses reward feedback on the duckietown gym for tracking the reward status.

Additionally, some other features has been hard coded:

1. The training images are stored as YUV color space, you can change it in line 258.

2. The frames are sized as 150x200, per original paper recommendation. This could be not the most effective resolution.

3. The logger resets if it detects driving out of bounds.

4) Automated log generation using pure pursuit

This baseline also provides an option to automatically generate training samples using the pure pursuit control algorithm.

The configurable parameters are similar to the human driver agent case described above.

If you would like to mass generate training samples on a headless server, under the `util` folder you will find the necessary tools.

To start pure pursuit data generation:

```
$ python3 automatic.py
```

5) Log using an actual Duckiebot

To log using an actual Duckiebot, refer to this tutorial on how to get a rosbag on a duckiebot.

Once you have obtained the ROS bag, you can use the folder `duckieRoad` to process that log.

6) Process a log from an actual Duckiebot

You will find the following files in the `duckieRoad` directory.

```
.
├── Dockerfile                        # File that sets up the docker image
|
├── bag_files                         # Put your ROS bags here.
|   ├── ROSBAG1                        # Your ROS bag.
|   ├── ROSBAG2                        # Your training on Date 2.
|   └── ...
|
├── converted                          # Stores the converted log for you
to train the Duckiebot
|
├── src                                # Scripts to convert ROS bag to
pickle log
|   ├── _loggers.py                    # Logger used to log the pickle log
|   ├── extract_data_functions.py      # Helper function for the script
|   └── extract_data.py                # Convertion script. You set your
Duckiebot
|                                        name, and topic to convert here.
|
├── MakeFile                           # Make file.
├── requirements.txt                   # Used for docker to setup dependen-
cy
└── pickle23.py                        # Convert the pickle2 style log pro-
duced to                                      pickle 3
```

https://docs.duckietown.org/daffy/duckietown-robotics-development/out/ros_logs.html

You should change `extract_data.py` line 83 to the correct `VEHICLE_NAME`.

First put your ROS bags in the bag_files folder. Then:

```
$ make make_extract_container
```

Next start the conversion docker:

```
$ make start_extract_data
```

It will automatically mount the bags folder as well as the converted folder.

NOTE: When you run the make file, make sure you are in duckieRoad not in the src folder!

3.4. The duckieLog

This folder is set for your to put all of your duckie logs. Some helper functions are provided. However, they might not be the most efficient ones to run. It is here for your reference.

1) The log viewer

To view the logs, under duckieLog folder:

```
$ python3 util/log_viewer.py --log_name YOUR_LOG_FILE_NAME.log
```

2) The log combiner

To combine the logs, under duckieLog folder:

```
$ python3 util/log_combiner.py --log1 dataset1.log --log2 dataset2.log
--output newdataset.log
```

3.5. The duckieTrainer

This section describes everything you need to know using the duckieChallenger.

1) Folder structure

In this folder you can find the following fils:

```
.
├── __pycache__                     # Python Compile stuff.
|
├── trainlogs                           # Training logs for tfboard.
|   ├── Date 1                      # Your training on Date 1.
|   ├── Date 2                      # Your training on Date 2.
|   └── ...
|
├── trainedModel                    # Your trained model is here.
|   ├── FrankNetBest_Loss.h5        # Lowest training loss model.
|   ├── FrankNetBest_Validation.h5  # Lowest validation loss model.
|   └── FrankNet.h5                 # The last model of the training.
|
├── frankModel.py                   # The deep learning model.
├── logReader.py                    # Helper file for reading the log
├── train.py                        # The training setup.
├── requirements.txt                # Required pip3 packges for training
└── train.log                       # Your training data.
```

2) Environment Setup

To setup your environment, I strongly urge you to train the model using a system with GPU. Tensorflow and GPU sometimes can be confusing, and I recommend you to refer to tensorflow documentation for detailed information.

Currently, the system requires `TensorFlow` 2.2.1. To setup TensorFlow, you can refer to the official TensorFlow install guide here.

Additionally, this training sytem utilizes `scikit-learn` and `numpy`. You can find a provided requirements.txt file that helps you install all the necessary packages.

```
$ pip3 install -r requirements.txt
```

3) Model Adjustment

To change the model, you can modify the `frankModel.py` file as it includes the model architecture. Currently it uses a parallel architecture to seperately generate a linear and angular velocity. It might perform better if they are not setup seperately.

To change your training parameters, you can find EPOCHS, LEARNING RATE, and BATCH size at the beginning of `train.py`. You should tweak around these values with respect to your own provided training data.

4) Before Training

Before you start training, make sure your log is stored at the root of the `duckieTrainer` folder. It should be named as `train.log`.

Make sure you have saved all the desired trained models into duckieModels. Trust me you do not want your overnight training overwritten by accident. Yes I have been through losing my overnight training result.

5) Train it

To train your model:

```
$ python3 train.py
```

To observe using tensorboard, run this command in the `duckieTrainer` directory:

```
$ tensorboard --logdir logs
```

You should be able to also see your training status at `http://localhost:6006/`. If your computer is accessible by other computers, you can also see it by visiting `http://TRAINERIP:6006`

6) Things to improve

There are a lot of things could be improved as this is an overnight hack for me. The data loading could be maybe more efficient. Currently it just load all and stores all in a global variable. The training loss reference might not be the best. The optimizeer might be improved. And most importantly, the way of choosing which model to use could be drastically improved.

7) Troubeshooting

```
Symptom: tensorflow.python.framework.errors_impl.InternalError: CUDA run-
time implicit initialization on GPU:0 failed. Status: out of memory

Resolution: Currently there is no known fix other than cross your fingers
and run again and reducing your batch size.
```

3.6. The duckieModels

This is a folder created just for you to keep track of all your potential models. There is nothing functional in it.

3.7. The duckieChallenger

This is the folder where you submit to challenge. The folder is structured as follows:

```
.
├── Dockerfile                      # Docker file used for compiling a
container.
|                                      Modify this file if you added file,
etc.
├── helperFncs.py                    # Helper file for all helper func-
tions.
├── requirements.txt                 # All required pip3 install.
├── solution.py                      # Your actual solution
└── submission.yaml                  # Submission configuration.
```

After you put your trained model `FrankNet.h5` in this folder, you can proceed as normal submission:

```
$ dts challenges submit
```

Or run locally:

```
$ dts challenges evaluate
```

An example submission looks like this

3.8. Acknowledgement

We would like to thank: Anthony Courchesne and Kay (Kaiyi) Chen for their help and support during the development of this baseline.

UNIT E-4
Dataset Aggregation

This section describes the procedure for training and testing an agent with the gym-duckietown simulator using the Dagger algorithm.

It can be used as a starting point for any of the `LF`, `LFV`, and `LFI` challenges.

KNOWLEDGE AND ACTIVITY GRAPH

Requires: You are somewhat familiar with PyTorch and the Pytorch template.

Results: You could win the AI-DO!

AI DO
The AI Driving Olympics
@ NeurIPS 2020
NEURAL INFORMATION PROCESSING SYSTEMS
aido.duckietown.org
& many supportive sponsors:
DUCKIETOWN
Motional
Swiss Re
NVIDIA
Today's Presenters: Chude Qian (U. Toronto), Moustafa ElaRaby (U. Montréal)

Figure 4.1. Dataset Aggregation (skip to end)

4.1. Introduction

We saw a first implementation of imitation learning in the behaviour cloning baseline. That baseline models the driving task as an end-to-end supervised learning problem where data can be collected offline from an expert. One of the central issues with this approach is that of **distributional shift**. Since this is a sequential decision making problem, the training data are not “identically and independently distributed”. The result is that if your agent deviates from the *optimal* trajectory that was demonstrated by the expert, it will not have any data in its dataset that shows it how to *recover back* to the optimal trajectory. As a result, it is unlikely that the behiaviour cloning approach will be robust.

For a better result than behaviour cloning this second version of imitation learning does not train only on a single trajectory given by the expert. We follow the Dataset Aggregation algorithm (Dagger) where we also let the agent interact with the environment and allow the expert to *recover*. The actions between the expert and the learner are chosen randomly with a varying probability with the hope that the expert *corrects* the learner if it starts deviating from the optimal trajectory.

4.2. Quickstart

Clone this repo:

```
$ git clone https://github.com/duckietown/challenge-aido_LF-baseline-
dagger-pytorch.git
```

Change into the directory:

```
$ cd challenge-aido_LF-baseline-dagger-pytorch
```

In here you will see two directories `submission` and `learning`. To make a submission, enter the `submission` folder:

```
$ cd submission
```

Then test the submission, either locally with:

```
$ dts challenges evaluate --challenge CHALLENGE_NAME
```

or make an official submission when you are ready with

```
$ dts challenges submit CHALLENGE_NAME
```

You can find the list of challenges here. Make sure that it is marked as "Open".

4.3. Local Development Workflow

The previous submission used a model which is included in the repo, but you should try to improve upon it.

1) Option 1: Training with Collab

We provide a Collab notebook that you can used to get started

During training the loss curve for each episode is available (by default on a folder created on root called `iil_baseline`) and may be checked using `tensorboard` and specifying the `--logidr`. On the same folder you will have `data.dat` and `target.dat` which are the memory maps used by the dataset.

2) Option 2: Training Locally

Start by cloning the gym-duckietown simulator repo:

```
$ git clone https://github.com/duckietown/gym-duckietown.git
```

Change into the directory:

```
$ cd gym-duckietown
```

Install the package:

```
$ pip3 install -e .
```

To run the baseline training procedure, run:

```
$ python -m learning.train
```

in the root directory.

3) Parameters that can affect training

There are several optional flags that may be used to modify hyperparameters of the algorithm:

- `--episode` or `-i` an integer specifying the number of episodes to train the agent, defaults to 10.
- `--horizon` or `-r` an integer specifying the length of the horizon in each episode, defaults to 64.
- `--learning-rate` or `-l` integer specifying the index from the list [1e-1, 1e-2, 1e-3, 1e-4, 1e-5] to select the learning rate, defaults to 2.
- `--decay` or `-d` integer specifying the index from the list [0.5, 0.6, 0.7, 0.8, 0.85, 0.9, 0.95] to select the initial probability to choose the teacher, the learner.
- `--save-path` or `-s` string specifying the path where to save the trained model, models will be overwritten to keep latest episode, defaults to a file named iil_baseline.pt on the project root.
- `--map-name` or `-m` string specifying which map to use for training, defaults to loop_empty.
- `--num-outputs` integer specifying the number of outputs the model will have, can be modified to train only angular speed, defaults to 2 for both linear and angular speed.
- `--domain-rand` or `-dr` a flag to enable domain randomization for the transferability to real world from simulation.
- `--randomize-map` or `-rm` a flag to randomize training maps on reset.

The baseline model is based on the Dronet model. The feature extractor of the model is frozen while the classifier is modified for the regression task.

All the PyTorch boilerplate code is encapsulated in the `NeuralNetworkPolicy` class implemented on `learning/imitation/iil-dagger/learner/neural_network_policy.py` and is based on previous work done by Manfred Díaz on Tensorflow.

4) Local Evaluation

A simple testing script `test.py` is provided with this implementation. It loads the latest model from the the provided directory and runs it on the simulator. To test the model:

```
$ python -m learning.test --model-path path
```

The model path flag has to be provided for the script to load the model:

- `--model-path` or `-mp` string specifying the path to the saved model to be used in testing.

Other optional flags that may be used are:

- `--episode` or `-i` an integer specifying the number of episodes to test the agent, defaults to 10.
- `--horizon` or `-r` an integer specifying the length of the horizon in each episode, defaults to 64.
- `--save-path` or `-s` string specifying the path where to save the trained model, models will be overwritten to keep latest episode, defaults to a file named iil_baseline.pt on the project root.
- `--num-outputs` integer specifying the number of outputs the model has, defaults to 2.
- `--map-name` or `-m` string specifying which map to use for training, defaults to loop_empty.

5) Expected Results

The following video shows the results for training the agent during 130 episodes and keeping the rest of the configuration to its default:

Figure 4.2

6) Tips to Improve your model

Some ideas on how to improve on the provided baseline:

- Map randomization.
- Domain randomization.
- Better selection than random when switching between expert/learner actions.
- Balancing the loss between going straight and turning.
- Change the task from linear and angular speed to left and right wheel velocities.
- Improving the teacher.

4.4. References

```
@phdthesis{diaz2018interactive,
  title={Interactive and Uncertainty-aware Imitation Learning: Theory
and Applications},
  author={Diaz Cabrera, Manfred Ramon},
  year={2018},
  school={Concordia University}
}

@inproceedings{ross2011reduction,
  title={A reduction of imitation learning and structured prediction to
no-regret online learning},
  author={Ross, St{\'e}phane and Gordon, Geoffrey and Bagnell, Drew},
  booktitle={Proceedings of the fourteenth international conference on
artificial intelligence and statistics},
  pages={627--635},
  year={2011}
}

@article{loquercio2018dronet,
  title={Dronet: Learning to fly by driving},
  author={Loquercio, Antonio and Maqueda, Ana I and Del-Blanco, Carlos
R and Scaramuzza, Davide},
  journal={IEEE Robotics and Automation Letters},
  volume={3},
  number={2},
  pages={1088--1095},
  year={2018},
  publisher={IEEE}
}
```

UNIT E-5

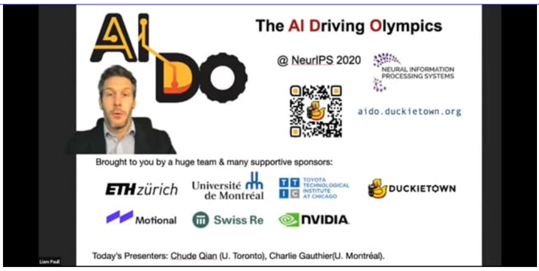
Residual Policy Learning

This section describes the basic procedure for making a submission with a model trained in simulation using residual policy learning with PyTorch and ROS. In this approach, we use the basic Duckietown lane following stack as the base policy, and we use reinforcement learning to improve it.

KNOWLEDGE AND ACTIVITY GRAPH

Requires: That you have made a submission with the ROS template.

Results: You have a submission that leverages both our ROS stack and reinforcement learning.



The AI Driving Olympics

@ NeurIPS 2020

aido.duckietown.org

Brought to you by a huge team & many supportive sponsors:

Today's Presenters: Chude Qian (U. Toronto), Charlie Gauthier(U. Montréal).

Figure 5.1. Residual Policy Learning

Before getting started, you should be aware that this baseline is a combination of the RL baseline and of the ROS template. It is recommended that you are familar for each of those templates and baselines, as the workflow of this one is similar to those. Here are some links:

- RL baseline
- ROS template
- Classical Duckietown baseline

You should also make sure you have access to good hardware. A recent graphics card (probably GTX1060+) is a must, and more than 8GB of RAM is required.

5.1. Quickstart

To train a policy, you should first make sure that Docker on your machine can access the GPU/CUDA. You should also install CUDA10.2+ locally.

Here's a few pointers:

- nvidia-docker
- CUDA 11

Clone this repo:

```
$ git clone  https://github.com/duckietown/challenge-aido_LF-baseline-
RPL-ros.git
```

Change into the directory:

```
$ cd challenge-aido_LF-baseline-RPL-ros
```

Test the submission, either locally with:

```
$ dts challenges evaluate --challenge CHALLENGE_NAME
```

or make an official submission when you are ready with

```
$ dts challenges submit --challenge CHALLENGE_NAME
```

You can find the list of challenges here. Make sure that it is marked as “Open”.

5.2. Baseline Overview

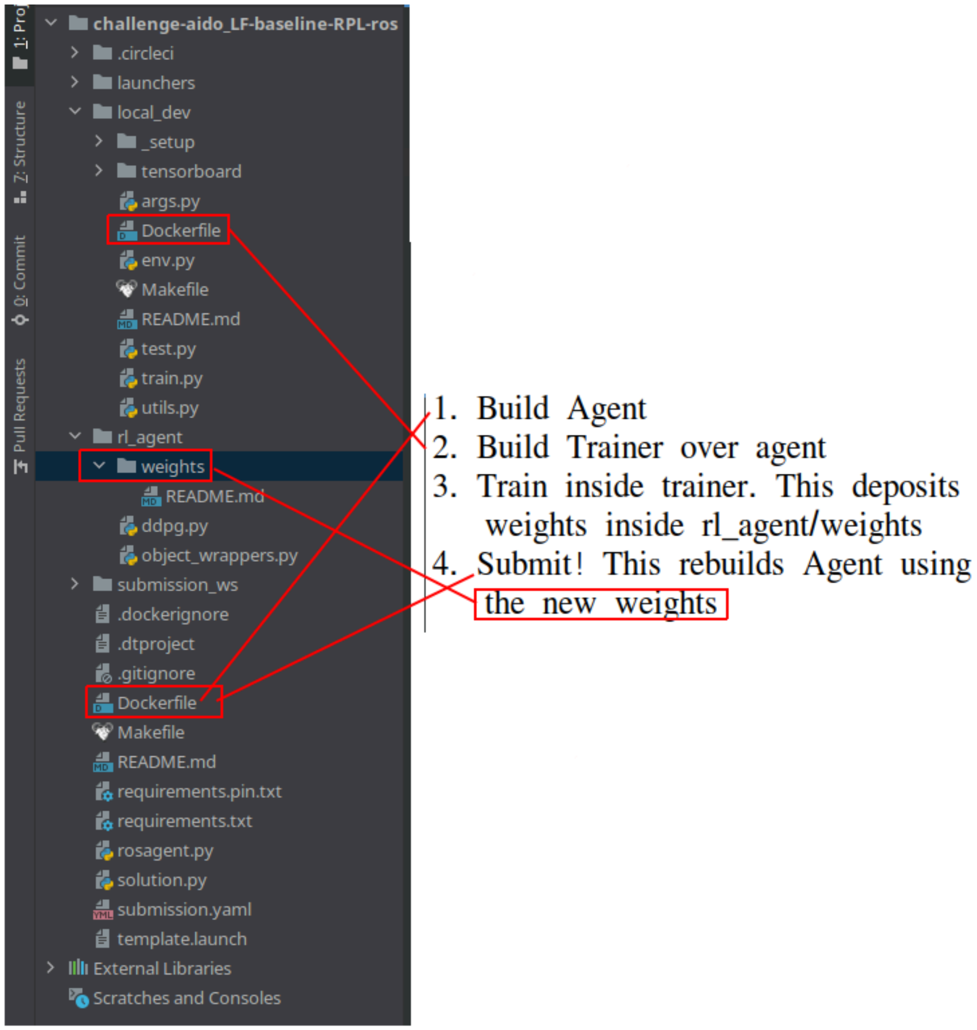
Since, this baseline uses both ROS and ML, we need to train inside an environment where both ROS and PyTorch are installed. We will use Docker for this purpose.

The ROS template already provides us with a submission docker image. Our strategy here is to directly use that agent docker image during training, but we’ll the addition of the simulator and the training architecture on top.

This could have been done using a second running docker container to provide a network interface to the simulator, but this adds unnecessary overhead since we don’t actually need the added security that comes with running things separately.

So, every time we train, we build the agent docker image, and then the “trainer docker” image builds directly `FROM` the agent image, adding the simulator on top.

The final docker container then runs the simulator and the agent in parallel, allowing the agent to directly interface with the simulator, just like we do in the other machine learning baselines.



1. Build Agent
2. Build Trainer over agent
3. Train inside trainer. This deposits weights inside rl_agent/weights
4. Submit! This rebuilds Agent using the new weights

Figure 5.2. RPL baseline overview

5.3. How to train your policy

From the `challenge-aido_LF-baseline-RPL-ros` directory, change into the `local_dev` directory:

```
$ cd local_dev
```

and open the `args.py` file. This is how you will control the training and testing in this repo. For now, just change the `--test` argument to `default=False`. Then, we can train with:

```
$ make run
```

As mentioned Section 5.2 - Baseline Overview, this will first build two subsequent docker images. This might take a while. Then, it will train an RL policy over the ROS stack inside Docker.

When it finishes, see how it works. Simply change the `--test` flag back to `default=True` in `args.py` and test with:

```
$ make run
```

This will launch a simulator window on your host machine for you to view how your agent performs. You should see something like this:

Figure 5.3

You can use this gif to gauge how long it takes for the testing docker to start (do note that this assumes that the two required docker images have already been built!)

5.4. How to submit the rained policy

Make sure that `rosagent.py` uses the right weights for your RL agent. This is controlled by the `MODEL_NAME` global variable. Then follow the procedure in Section 5.1 - Quickstart to evaluate and submit.

5.5. How to improve your policy

First, you should probably improve the base ROS policy. By default, this baseline uses the basic `lane_following` demo that is provided in Duckietown (unknown ref opmanual_duckiebot/demo-lane-following)

previous **warning** (18 of 18) index
warning

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

 > I do not know what is indicated by the link '#opmanual_duckiebot/demo-lane-following'.
```

Location not known more precisely.

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

.

You could build a Pure Pursuit controller, change the lane filter, etc. See the classical Duckietown baseline for more ideas. To do this, you would add your new ROS packages inside of `submission_ws`.

You could also limit RL's influence over the final policy. Perhaps the current approach of giving it full control in [-1,1] action values isn't restrictive enough. Perhaps it could be better if it could only change the base policy by smaller action values.

Or perhaps it's the opposite: maybe the base policy needs to be changed by more than `1`: since the min/max value that the base policy can output is `1`/`-1`, the RL policy would need to be able to output from `-2` to `2` to fully correct it.

Here are some ideas for improving your policy:

- Check out the `dtRewardWrapper` in `rl_agent` and modify the rewards (set them higher or lower and see what happens). By default, this wrapper is not used: you will have to add it to `train.py`.
- Try resizing the images. Make them smaller to have faster training, or bigger for making sure that RL can extract everything it can from them. You will need to also edit the layers in `ddpg.py` accordingly.
- Try making the observation image grayscale instead of color.
- Try stacking multiple images, like 4 monochrome images instead of 1 color image. You will need to also edit the layers in `ddpg.py` accordingly.
- You can also try increasing the contrast in the input to make the difference between road and road-signs clearer. You can do so by adding another observation wrapper.
- Cut off the horizon from the image (and correspondingly change the convnet parameters).
- Check out the default hyperparameters in `local_dev/args.py` and tune them. For example increase the `expl_noise` or increase the `start_timesteps` to get better exploration.
- (more sophisticated) Use a different map in the simulator, or - even better - use randomized maps. But be mindful that some maps include obstacles on the road, which might be counter-productive to a `LF` submission.
- (more advanced) Use a different/bigger convnet for your actor/critic. And add better initialization.
- (very advanced) Use the ground truth from the simulator to construct a better reward
- (extremely advanced) Use an entirely different training algorithm - like PPO, A2C, or DQN. Go nuts. But this might take significant time, even if you're familiar with the matter.

5.6. Sim2Real Transfer (Optional)

You should try your agent on the real Duckiebot.

It is possible, even likely, that your agent will not generalize well to the real environment. One approach to mitigate this problem is to randomize the simulator environment during training, in the hope that this improves generalization. This approach is referred to as "Domain Randomization".

To implement this, you will need to modify the `local_dev/env.py` file. You'll notice that we launch the `Simulator` class from `gym-duckietown`. When we take a look at the constructor, you'll notice that we aren't using all of the parameters listed. In particular, the three you should focus on are:

- `map_name`: What map to use; hint, take a look at gym_duckietown/maps for more choices
- `domain_rand`: Applies domain randomization, a popular, black-box, sim2real tech-

nique

- `randomized_maps_on_reset`: Slows training time, but increases training variety.

Mixing and matching different values for these will help you improve your training diversity, and thereby improving your evaluation robustness!

PART F

Reference manual

We have built some tools and infrastructure to make it easy to build solutions. These tools may be helpful in building an efficient workflow for developing and testing your solutions before you submit them.

Contents

Unit F-1
`dts challenges` CLI

This section is a reference for how to interact with the challenges server with the command line.

1.1. Account info

Use this command to see the status of your account:

```
$ dts challenges info
```

1.2. Local evaluation

The `evaluate` command allows you to do a local evaluation of your submission:

```
$ dts challenges evaluate
```

1.3. Submitting a submission

The `submit` command allows you to submit the solution in the current directory:

```
$ dts challenges submit
```

There are many options for this command, explained in Unit F-3 - Advanced submission options.

1.4. List submissions

The `list` command allows you to see all of your submissions:

```
$ dts challenges list
```

1.5. Reset a submission

Resetting a submission means that you discard the evaluations already perfomed and you force them to be done again.

```
$ dts challenges reset --submission ID
```

1.6. Retire a submission

Retiring a submission means that you declare the submission void. It will not be evaluated and previous results will be discarded.

```
$ dts challenges retire --submission ID
```

1.7. Follow the fate of a submission

The `follow` command polls the server to see whether there are updates:

```
$ dts challenges follow --submission ID
```

1.8. Defining a challenge

The `define` command allows to *define* a challenge:

```
$ dts challenges define
```

UNIT F-2
Using the Evaluator

This section describes how to use the Challenges evaluators.

2.1. Evaluators

An evaluator is a machine that is in charge of evaluating the protocols.

2.2. Running your own evaluator

We have several evaluators online that process jobs.

If you want to avoid waiting in the queue for to long, you can run your own evaluator.

The command line is:

```
$ dts challenges evaluator --continuous
```

This evaluator will connect to the server and **execute preferentially your submissions.**

2.3. Advanced options for evaluator

1) Naming evaluator

Use the option `--name` to name the evaluator instance:

```
$ dts challenges evaluator --name a name
```

Otherwise the name is going to be autogenerated.

For example:

```
$ dts challenges evaluator --name Instance1 &
$ dts challenges evaluator --name Instance2 &
```

2) Run a specific submission

Run the evaluator on a specific submission:

```
$ dts challenges evaluator --submission ID
```

This evaluates a specific submission.

Note that to force re-evaluation of a submission, you must first reset the submission.

Also note that you cannot re-evaluate a submission that has been “retired”.

UNIT F-3
Advanced submission options

This section describes additional options for the `dts challenges submit` command.

3.1. `submission.yaml` file

Each submission directory has a file `submission.yaml` containing the following information:

```
protocol: protocol # do not change
challenge: challenge name(s)
user-label: optional label
user-payload: optional user payload
```

You can override these using the command line, as explained below.

3.2. Specifying the challenge

However you can also pass the name as a parameter `--challenge`:

```
$ dts challenges submit --challenge challenge name
```

The names of the challenges can be seen at this page.

For example, if you would only like to submit to submit to LF validation system, you can do it as:

```
$ dts challenges submit --challenge aido3-LF-sim-validation
```

If you would like to submit to multiple specific challenges, you can do it in the yaml file:

```
protocol: aido2_db18_agent-z2 # do not change
challenge: [challenge1_name,challenge2name,...]
```

3.3. Metadata

You can attach two pieces of metadata to your submission.

1. A human-readable label for your identification.
2. A small JSON payload that describes the details of your submission, such as the parameters that you used for your algorithm.

To specify the label, use the option `--user-label`:

```
$ dts challenges submit --user-label "My label"
```

To specify the payload, use the option `--user-meta` and specify a JSON structure:

```
$ dts challenges submit --user-meta '{"param":"1"}
```

3.4. Skip Docker cache

Use the option `--no-cache` to avoid using the Docker cache and re-build your containers from scratch:

```
$ dts challenges submit --no-cache
```

Part G
References

[1] Edwin Olson. AprilTag: A robust and flexible visual fiducial system. In *IEEE International Conference on Robotics and Automation (ICRA)*, pages 3400–3407, 2011.
[2] Daniel Pickem, Paul Glotfelter, Li Wang, Mark Mote, Aaron Ames, Eric Feron, and Magnus Egerstedt. The Robotarium: A remotely accessible swarm robotics research testbed. In *Robotics and Automation (ICRA), 2017 IEEE International Conference on*, pages 1699–1706. IEEE, 2017.