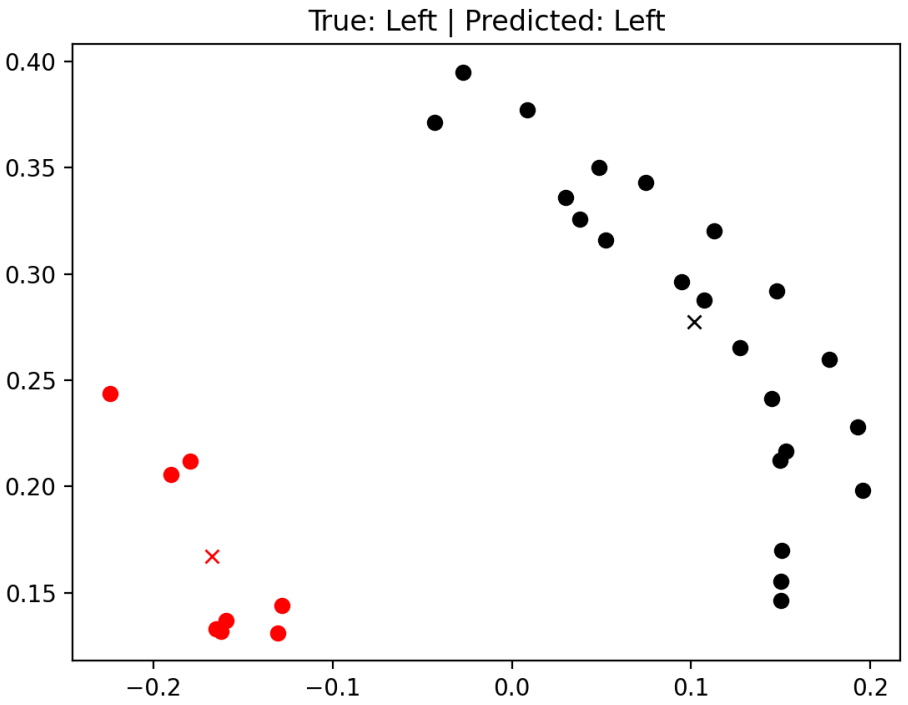
UNIT ZERO-COUNTER-1

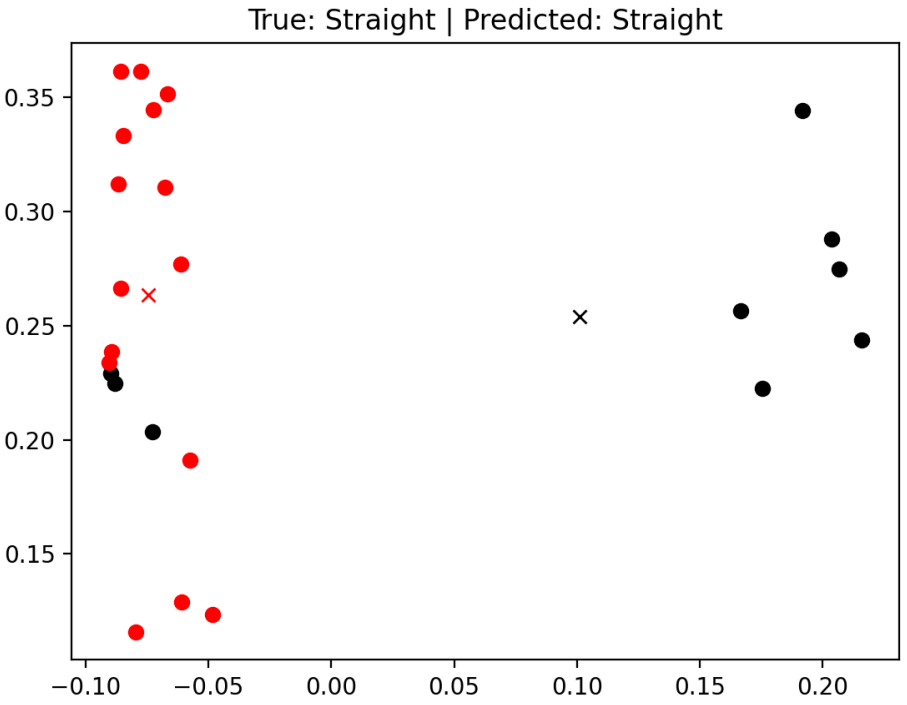
Lisus: Assignment 3

1.1. The Final Result

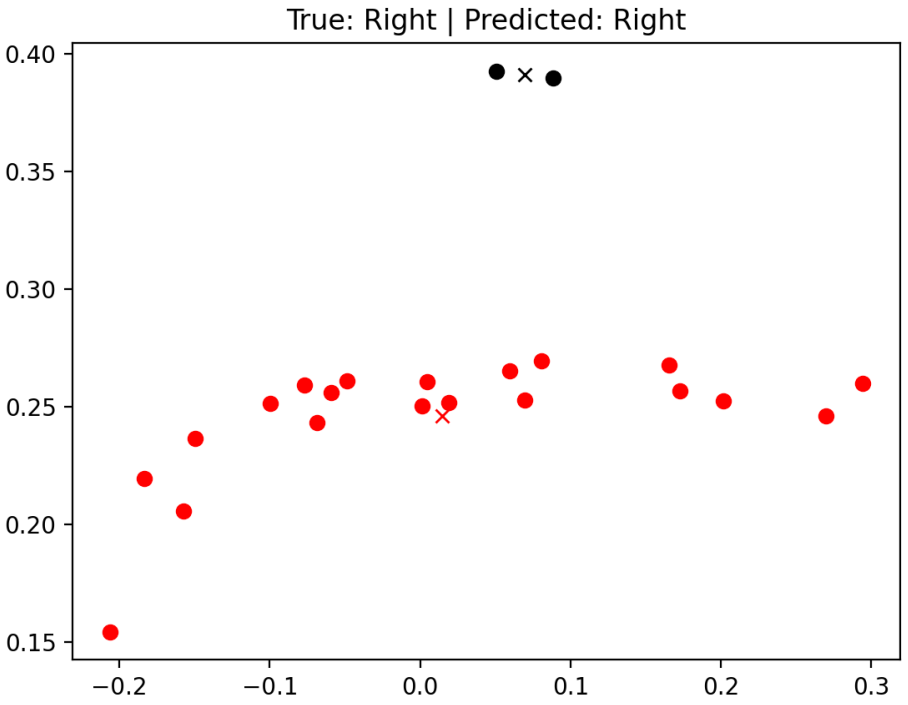
The goal of this assignment was to train a Gaussian Process (GP) model to predict the state of the lane that the Duckiebot is currently in. The final results of the GP performance is shown below. The visualization shows the detected white (in black) and yellow (in red) lane segments with an x marking the average position of each type of segments. The true and predicted lane state is listed in the title. Please refer to instructions to reproduce to reproduce these results.



(a) Left state



(b) Straight state



(c) Right state

Figure 1.1. Examples of succesfully predicted states.

1.2. Mission and Scope

The mission was to take the detected white and yellow line segments and to use them as inputs to predict whether the current lane is in a “straight”, “left”, or “right” portion.

1) Motivation

A large number of control algorithms could greatly benefit from knowing which part of the lane the robot is in. In fact, I used a more “hacky” estimator, described in the next section, for both Assignment 1 and Assignment 2.

In Assignment 1, we were tasked with coding up a simple pure-pursuit controller. My solution relied on finding the white and yellow lanes in front of the robot, and then placing a target point in between the two lanes at some desired distance. Issues arose when only one of the lanes was detected. When this happened, I could blindly place the target point at half the lane width to the left of the white line or at half the lane width to the right of the yellow line. However, this would work only in a straight portion of the lane, as during turns the lanes could be almost entirely horizontal and placing a target point to the left or the right would yield a target that was on the lane itself. As such, when I was able to detect that I was in a left turn for example, I could instead place the target below the detected white line instead of the left of it, yielding much better per-

formance.

In Assignment 2, I used a simple PID controller with an estimator of the robots pose in the lane in order to achieve lane following. I wanted the robot to go as fast as possible, however this frequently yielded problems with the bot flying off the road during turns. As such, knowing the state of the lane allowed me to do simple gain scheduling which made the robot be able to turn sharper and drive slower only during turning portions.

In both of these cases, the robustness and performance of my algorithms was greatly improved by incorporating knowledge about whether the lane was going straight, turning left or turning right.

2) Existing solution

There has not previously been an explicit lane state estimator which estimated whether the lane was going straight or turning left or right. I coded up such an estimator for Assignment 1 and used it also in Assignment 2. However, this previous estimator was simply a large number of "if-else" statements and thus very bug-prone and not robust.

3) Opportunity

Although I coded up a functional "brute-force" lane state detector, it was extremely error prone during non-standard line detection situations. As such, I would frequently have to figure out what cases I missed and include additional checks and balances in my code to accommodate them.

Instead of doing this, I thought that a simple learning algorithm such as a Gaussian Process would be well suited to tackle this problem. With enough training samples, the algorithm would in theory learn to recognize essentially all possible line detection outliers and incorporate them into the prediction. This would not only improve the robustness and performance of the algorithm, but also greatly decrease the amount of code required to be run and debugged in algorithms that make use of the lane state detection prediction.

1.3. Background and Preliminaries

The only background required above the basic understanding of the functionalities of the duckiebot and line detections is knowing what a Gaussian Process (GP) is. For this problem, I used the publicly available *scikit learn GaussianProcessClassifier* python library to do the hard work for me. As such, it is not necessary to understand the in-depth math behind GPs, and the reader is heavily referred to the informally called *Gaussian Process Bible* by Rasmussen and Williams which summarizes Gaussian Processes in-depth (actual reference: *Carl Edward Rasmussen and Christopher K. I. Williams. 2005. Gaussian Processes for Machine Learning (Adaptive Computation and Machine Learning). The MIT Press.*). However, below is a quick overview of the method.

Specifically, this solution makes use of Gaussian Processes for Classification, as opposed to the arguably more commonly used Regression. In the Classification task, we aim to find some functional relationship between inputs $\mathbf{x}$ and an output $\boldsymbol{y}$, which belongs to some list of categories $\mathbf{y}$. For our situation, the $\mathbf{x}$ is the properties of the currently seen white and yellow segments (such as their average position) and $\mathbf{y}$ is simply

the options “straight”, “left”, and “right”. Effectively, for some segment information input, we want the algorithm to predict which of the options is most likely to correspond to this input. GPs use a much simpler approach than other machine learning methods and simply compare newly provided inputs to previously known input-output relations in order to predict a new output. As such, training involves learning the parameters of the function, called the kernel, which evaluates this “closeness” between new and known inputs. Once these parameters are learned, the GP is able to use this kernel to evaluate new inputs and predict an output based on how closely related the new input is to known inputs during training. As a downside, the GP approach does require the user to keep track of all inputs used during training. For this problem however, this does not prove to be a barrier as the inputs are very small and the simplicity of the problem requires only a couple hundred training samples.

1.4. Definition of the problem

Up to now it was all fun and giggles. This is the most important part of your report: a crisp, possibly mathematical, definition of the problem you tackled. You can use part of the preliminary design document to fill this section.

Final Objectives/Deliverables

- Predict whether the lane portion the bot is currently in is going “straight”, “left”, or “right” based on detected yellow and white lane segments. This prediction should be tested in simulator and on the real robot.
- Produce a method to collect training data for the mission.
- Produce a trained model which can be loaded directly into a control algorithm.

Assumptions:

- The algorithm is not expected to function when the robot is not between a yellow and white line.
- The algorithm is not expected to function in situations where the robot is not decently positioned even when it is between a yellow and white line. Examples include
- The robot is more than ~45 degrees declined from the correct flow of traffic in a lane, and
- The robot is backwards with the yellow line on the right and the white line on the left.

Performance Metrics

Performance of the Gaussian Process algorithm will be evaluated on a collected test set of both simulated and real world examples. This test set will have user specified true lane states and these will be compared against the lane states predicted by the algorithm to calculate a percent success rate. This success rate will be evaluated for only simulated results, only real world results, and for a combination of the two.

As a more qualitative metric, the GP will be implemented in a pure-pursuit algorithm in order to verify its ability to predict the lane state in an online manner. Since the lane state prediction will only be indirectly measured by the performance of the controller, no numerical conclusions will be made from this.

1.5. Contribution / Added functionality

Describe here, in technical detail, what you have done. Make sure you include: - a theoretical description of the algorithm(s) you implemented - logical architecture - software architecture - details on the actual implementation where relevant (how does the implementation differ from the theory?) - any infrastructure you had to develop in order to implement your algorithm - If you have collected a number of logs, add link to where you stored them

Feel free to create subsections when useful to ease the flow

Data Recording

The first step to accomplish this project was the develop data recording infrastructure. For a given recording, I needed to save the

- White lane segments + stats (such as average point, closest point, furthest point)
- Yellow lane segments + stats (such as average point, closest point, furthest point)
- The user-specified state of the lane

The exact procedure to record new data is described in the instructions to reproduce, however it should be very easy to expand this procedure to record other data if a more complex machine learning approach would want to be tried. The collected data is saved as a JSON file and the collected JSON files from the simulator and real world can be accessed at https://github.com/lisusdaniil/dt-exercises/tree/daffy/lane_state_prediction/exercise_ws/data.

GP Training

The next step in the development process and also another stand-alone deliverable is the actual training runfile. This file loads in any number of training and testing JSON files collected using the method listed in Data Recording above and subsequently trains and tests a GP model. It also provides an option to save the trained model using the *pickle* python library and also provides an option to load in a pickled model instead of training a new one.

Additionally, this function is also capable of augmenting the data. This is done using two approaches:

1. For recordings where both white and yellow segments are available, new training points are created using only the white and only the yellow lines with the same corresponding lane state. Thus, for every training point which has both white and yellow segments, 2 additional augmented training points are possible.

2. For data recorded during turning lane states, augmented training points can be generated by simply flipping the horizontal coordinates of the segments and changing the turn type. This is possible since the segments for a left turn and the segments for a right turn are simply horizontally flipped versions of each other. As such, for every training point corresponding to a turn, 1 additional augmented training point is generated.

Although data augmentation is implemented, it was not found to produce any additional benefits and thus was not used for training. Potentially in a more complex scenario it could be useful however.

The training runfile can be accessed at https://github.com/lisusdaniil/dt-exercises/blob/daffy/lane_state_prediction/train_GP.py.

GP Prediction in Unrelated Algorithm

Finally, an implementation of the trained GP model in a control algorithm is also provided. The algorithm is the same pure pursuit algorithm as the one described in the motivation section. The previous brute-force method has been switched out for the GP implementation for predicting the lane state. Although the controller node used in the code also contains the data collection functions for the sake of this assignment, the inclusion of the GP to predict the lane state is completely independent from the data collection. As such, by simply including a subsriber to the */ground_projection_node/lineseglist_out* topic and doing some segment pre-processing it is possible to include the GP prediction in any algorithm! The function triggered through the line */ground_projection_node/lineseglist_out* topic does the pre-processing and calls a saved GP model to produce a predicted direction. Depending on the need, a user can either create a new publisher or simply use the direction in the current node as was done for this implementation. The reader is again referred to the more detailed instructions to reproduce.

The full folder implemented using the *dt-exercises* approach can be found at https://github.com/lisusdaniil/dt-exercises/tree/daffy/lane_state_prediction.

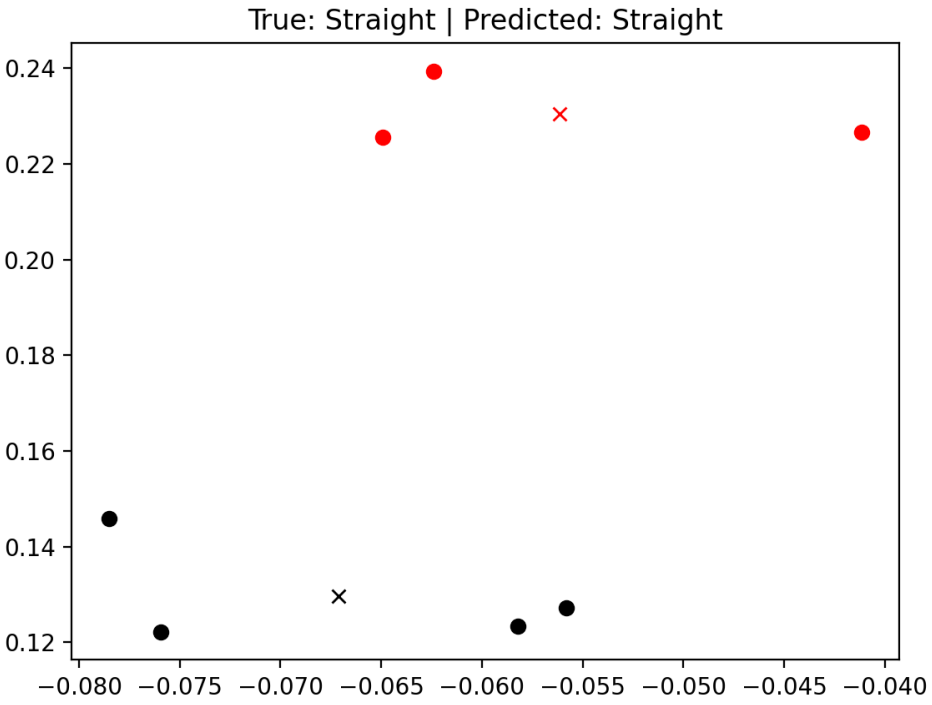
1.6. Formal performance evaluation / Results

As stated in the performance metrics section, qualitative metrics were evaluated on the GP performance using a direct testing set of simulated and real world collected datasets. Note that the model itself was trained using data from both the simulator and the real world. The results are shown below

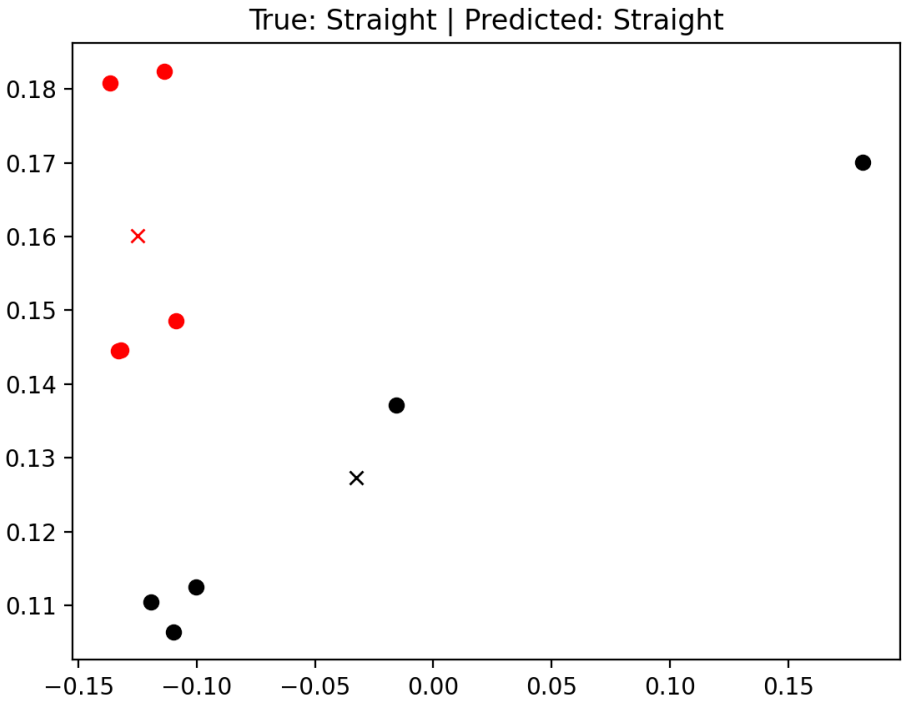
TABLE 1.1. RESULTS

Test Type	Accuracy
Sim	88-93%
Real	70-85%
Both	82-88%

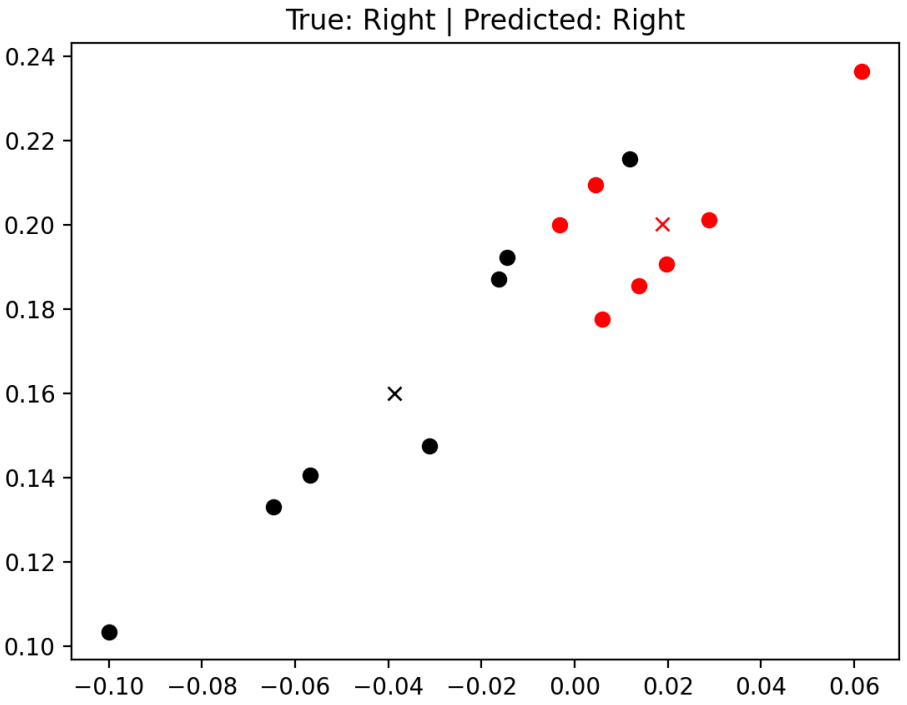
As can be seen, the algorithm performs quite well in both simulated and real world situations. Since GPs are stochastic processes, both the model training and model testing produce slightly different results depending on the given run. As such, several tests were run leading to the indicated ranges. It should be noted that the only reason that the real world performance is so much lower is due to the more unpredictable line detection. Particularly, my testing location was different from where I had previously calibrated my robots colour detection. However, even in these absurdly poor conditions, as can be seen in the noisy images below, the GP is capable of correctly predicting the majority of the results. As such, it is expected that with better calibrations the real world perfromance should be equivalent to that of the simulator.


True: Straight | Predicted: Straight
0.24
0.22
0.20
0.18
0.16
0.14
0.12
−0.080
−0.075
−0.070
−0.065
−0.060
−0.055
−0.050
−0.045
−0.040

(a)


True: Straight | Predicted: Straight
0.18
0.17
0.16
0.15
0.14
0.13
0.12
0.11
−0.15
−0.10
−0.05
0.00
0.05
0.10
0.15

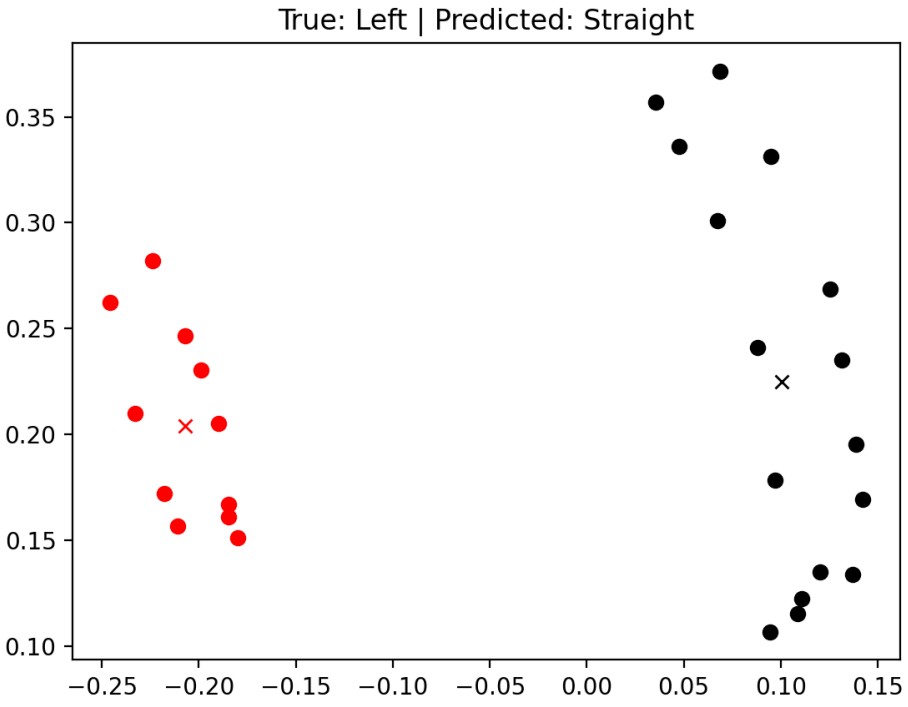
(b)



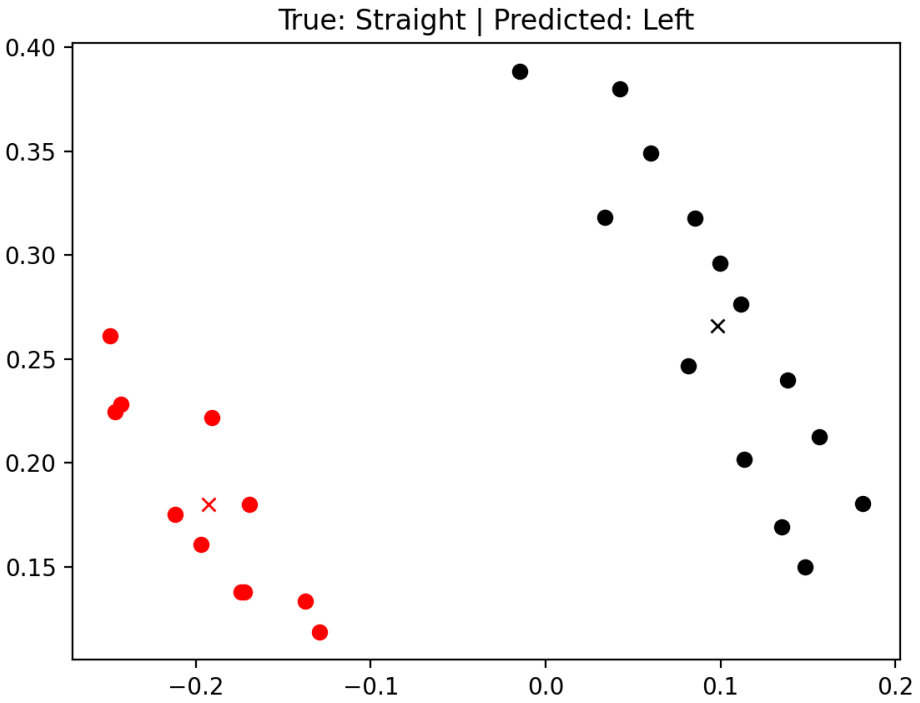
(c)

Figure 1.2. Examples of succesfully predicted noisy real-world segments.

Additionally, it should also be noted that many of the errors present in the testing set were arguably not errors from the perspective of our goal. As can be seen in the images below, in the True: Straight case where the GP predicts a left turn, a controller should probably be trying to more aggressively turn left given the clear deviation from the center. Likewise in the True: Left case, there does not appear a need to be rapidly turning left and straight controller gains would likely be preferred.


True: Left | Predicted: Straight
0.35
0.30
0.25
0.20
0.15
0.10
−0.25
−0.20
−0.15
−0.10
−0.05
0.00
0.05
0.10
0.15

(a)



(b)

Figure 1.3. Examples of incorrect simulator predictions.

1.7. Future avenues of development

Although the project meets the desired level of accuracy, there are a few additions which could be attempted. They are:

- Implement a proper validation set to ensure that the GP is being fully trained rather than the informal tests performed which amounted to "I am adding more training points with no visible benefits, therefore we are done."
- Train the GP using more than just the average, minimum and maximum values of each type of line segment.
- Train the GP to predict additional classes such as "approaching a left turn" and "approaching a right turn".
- Implement additional data augmentation such as randomly ignoring a certain number of segments and creating new training points from the rest.

UNIT ZERO-COUNTER-2

Lisus Assignment 3 Instructions

KNOWLEDGE AND ACTIVITY GRAPH

> **Requires:** Duckiebot in configuration `DB18` or `DB19`.
> **Requires:** Duckietown without intersections.
> **Requires:** Camera calibration completed.
> **Requires:** Wheel calibration completed.

2.1. Video of expected results

This video shows the Gaussian Process (GP) lane state estimate being used in a pure-pursuit controller. These instructions detail how to reproduce this result.

Figure 2.1. Expected pure pursuit results.

2.2. Laptop setup notes

The laptop should be pushed with the latest version of the `duckietown shell`. The full folder implemented using the *dt-exercises* approach can be found at https://github.com/lisusdaniil/dt-exercises/tree/daffy/lane_state_prediction. Simply clone the repo and launch `dts exercises test` with either the simulator or the duckiebot implementation.

2.3. Duckietown setup notes

This is a simple lane following algorithm so ideally the duckietown is set up in an infinite loop.

2.4. Pre-flight checklist

Check 1: The duckiebot is fully set up, with the user being able to control it through the joystick and being able to see its camera topics published in RVIZ.

Check 2: The `duckietown shell` and duckiebot have both been upgraded to the latest

version.

Check 3: The `lane_state_prediction` folder is cloned and `dts exercises build` has been run successfully.

Check 4: In order to train the GP model, the users personal machine must have the `json`, `numpy`, `sklearn`, `pickle`, and `matplotlib` Python libraries installed. This is because the GP training file is run outside of the duckietown shell!

2.5. Pure Pursuit Implementation Instructions

Below are instructions to reproduce the video shown above, which makes use of a trained GP model to predict the lane state in a pure pursuit controller. These instructions include changes needed to run the algorithm in the simulator vs on the duckiebot.

Step 1: Go to the `config/default.yaml` files in the `lane_controller_node` and in the `line_detector_node` and make sure that either the SIM or IRL settings are uncommented depending on whether you wish to run the test in the simulator or in the real world respectively. For the `line_detector_node`, it is recommended that the user tunes the colour settings to their own environment for best results.

Step 2: Either launch the simulator through `dts exercises test --sim` or launch the code on the duckiebot `dts exercises test -b` *DUCKIEBOT_NAME* `--local`.

Step 3: Open up the noVNC2 web app hosted by default at localhost:8087. Launch the virtual joystick app by running `dt-launcher-joystick` in the noVNC2 command prompt to control the robot. The robot's current view can be visualized in `RQT Image View` on topic `/agent/camera_node/image/compressed`. Additionally, the user can view the detected and projected line segments that are used in the GP prediction through the `RQT Image View` on topic `/agent/ground_projection_node/debug/ground_projection_image/compressed`.

Step 4: Use the joystick to ensure the duckiebot is positioned within the lane, ideally facing the correct flow of traffic head on.

Step 5: Simply press the `a` key while in the joystick to start the autonomous lane following. Exit lane following and control the bot manually at any point by pressing the `s` key.

Step 6: Regardless of whether the lane following is activated, the program will print the currently predicted lane state to the command prompt of your local machine from where the the exercise was launched from. Drive around manually and compare the output to what you think the lane state is. The provided trained GP model should produce an accuracy of approximately 90% in simulation and 80% in the real world.

2.6. Data Collection and Training Instructions

Below are instructions if the user wishes to collect more data and train their own model.

Step 1: Specify the desired file name to store your data in through the `train_filename_GP` parameter in the `lane_controller_node` `config/default.yaml` file. Please ensure that the file ends with `.json`.

Step 2: Follow Steps 1-3 from the pure pursuit instructions.

Step 3: Use the joystick to drive the duckiebot around to desired collection spots. Press the `e` key to trigger saving mode and then click the `up`, `left`, or `right` arrows to save the current line segments as corresponding to a straight, left, or right lane state respectively. As soon as a segment is saved, the terminal will display your selection and the saving mode will exit. If you wish to exit saving mode without saving anything, simply click the `s` key.

Step 4: When you have collected all of your desired data, press the `e` key and then the `down` arrow to save your data to the file name you specified in Step 1. This will not erase your current data, so you can continue adding to your current set after saving. The file will be saved to the `/code/exercise_ws/data/` directory.

Step 5: If you wish to restart collection or to save to a different file name, change the file name and restart your agent.

Step 6: When you have collected whatever datasets you wish, open the `train_GP.py` file and include the desired files in the `list_train_files` or `list_test_files`. The lists can take any number of seperate files. There are also tags to control whether to load the GP model from file or to train a new one, whether to save the model you'll use, and whether to do data augmentation. The parameters for the training have been chosen based on some trial and error, but the reader is referred to https://scikit-learn.org/stable/modules/generated/sklearn.gaussian_process.GaussianProcessClassifier.html for additional information.

Step 7: If `save_model` is set to `True`, the trained model will be saved in the `/code/exercise_ws/data/` directory under the name specified in `model_save_name`. If you wish to run the lane following with your new model, simply indicate the name of the model in the `GP_model_file` parameter in the `lane_controller_node` `config/default.yaml` file. Only the filename itself needs to be specified, as the code automatically calls the `/code/exercise_ws/data/` directory.

2.7. Troubleshooting

Symptom: The lane following fails miserably.

Resolution: Make sure to change the settings between SIM and IRL depending on whether you are running the simulator or the real world test. Additionally, feel free to play around with the `Controller settings` in the `lane_controller_node` `config/default.yaml` file.

Symptom: The lane prediction is always showing up as `Straight`.

Resolution: Make sure to set the `predict_lane_state` parameter to `True` in the `lane_controller_node` `config/default.yaml` file. If it is set to `False` the prediction will stay at the default lane state which is `Straight`.

Symptom: I cannot load in my saved GP model.

Resolution: Make sure you place your saved file in the `/code/exercise_ws/data/` directory.

Symptom: No matter what training files I input, my GP model preforms the same way.

Resolution: Make sure that the `load_model` parameter is set to `False` in the

`train_GP.py` file. Otherwise the trainer will just be loading in an existing saved GP model.

2.8. Demo failure demonstration

The controller could fail if the GP model makes incorrect predictions. The reader is referred to the results section for more details about the types of prediction failures and a discussion on their significance.

Part A
Caffe and Tensorflow

Contents

UNIT A-1

How to install PyTorch on the Duckiebot

PyTorch is a Python deep learning library that's currently gaining a lot of traction, because it's a lot easier to debug and prototype (compared to TensorFlow / Theano).

To install PyTorch on the Duckietbot you have to compile it from source, because there is no pro-compiled binary for ARMv7 / ARMhf available. This guide will walk you through the required steps.

1.1. Step 1: install dependencies and clone repository

First you need to install some additional packages. You might already have installed. If you do, that's not a problem.

```
sudo apt-get install libopenblas-dev cython libatlas-dev m4 libblas-dev
```

In your current shell add two flags for the compiler

```
export NO_CUDA=1 # this will disable CUDA components of PyTorch, be-
cause the little RaspberriPi doesn't have a GPU that supports CUDA
export NO_DISTRIBUTED=1 # for distributed computing
```

Then `cd` into a directory of your choice, like `cd ~/Downloads` or something like that and clone the PyTorch library.

```
git clone --recursive https://github.com/pytorch/pytorch
```

1.2. Step 2: Change swap size

When I was compiling the library I ran out of SWAP space (which is 500MB by default). I was successful in compiling it with 2GB of SWAP space. Here is how you can increase the SWAP (only for compilation - later we will switch back to 500MB).

Create the swap file of 2GB

```
sudo dd if=/dev/zero of=/swap1 bs=1M count=2048
```

Make this empty file into a swap-compatible file

```
sudo mkswap /swap1
```

Then disable the old swap space and enable the new one

```
sudo nano /etc/fstab
```

This above command will open a text editor on your `/etc/fstab` file. The file should have this as the last line: `/swap0 swap swap`. In this line, please change the `/swap0` to `/swap1`. Then save the file with CTRL+o and ENTER. Close the editor with CTRL+x.

Now your system knows about the new swap space, and it will change it upon reboot, but if you want to use it right now, without reboot, you can manually turn off and empty the old swap space and enable the new one:

```
sudo swapoff /swap0
sudo swapon /swap1
```

1.3. Step 3: compile PyTorch

`cd` into the main directory, that you clones PyTorch into, in my case `cd ~/Downloads/pytorch` and start the compilation process:

```
python setup.py build
```

This shouldn't create any errors but it took me about an hour. If it does throw some exceptions, please let me know.

When it's done, you can install the pytorch package system-wide with

```
sudo -E python setup.py install # the -E is important
```

For some reason on my machine this caused recompilation of a few packages. So this might again take some time (but should be significantly less).

1.4. Step 4: try it out

If all of the above went through without any issues, congratulations. :) You should now have a working PyTorch installation. You can try it out like this.

First you need to change out of the installation directory (**this is important - otherwise you get a really weird error**):

```
cd ~
```

Then run Python:

```
python
```

And in the Python interpreter try this:

```
>>> import torch
>>> x = torch.rand(5, 3)
>>> print(x)
```

1.5. (Step 5, optional: unswap the swap)

Now if you like having 2GB of SWAP space (additional RAM basically, but a lot slower than your built-in RAM), then you are done. The downside is that you might run out of space later on. If you want to revert back to your old 500MB swap file then do the following:

Open the `/etc/fstab` file in the editor:

```
sudo nano /etc/fstab
```

please change the `/swap0` to `/swap1`. Then save the file with CTRL+o and ENTER. Close the editor with CTRL+x.

UNIT A-2

How to install Caffe and Tensorflow on the Duckiebot

Caffe and TensorFlow are popular deep learning libraries, and are supported by the Intel Neural Computing Stick (NCS).

2.1. Caffe

1) Step 1: install dependencies and clone repository

Install some of the dependencies first. The last command "sudo pip install ....." will cause some time.

```
sudo apt-get install -y gfortran cython
sudo apt-get install -y libprotobuf-dev libleveldb-dev libsnappy-dev li-
bopencv-dev libhdf5-serial-dev protobuf-compiler git
sudo apt-get install --no-install-recommends libboost-all-dev
sudo apt-get install -y python-dev libgflags-dev libgoogle-glog-dev li-
blmdb-dev libatlas-base-dev python-skimage
sudo pip install pyzmq jsonschema pillow numpy scipy ipython jupyter
pyyaml
```

Then, you need to clone the repo of caffe

```
cd
git clone https://github.com/BVLC/caffe
```

2) Step 2: compile Caffe

Before compile Caffe, you have to modify Makefile.config

```
cd caffe
cp Makefile.config.example Makefile.config
sudo vim Makefile.config
```

Then, change four lines from

```
'#'CPU_ONLY := 1
/usr/lib/python2.7/dist-packages/numpy/core/include
INCLUDE_DIRS := $(PYTHON_INCLUDE) /usr/local/include
LIBRARY_DIRS := $(PYTHON_LIB) /usr/local/lib /usr/lib
```

to

```
CPU_ONLY := 1
/usr/local/lib/python2.7/dist-packages/numpy/core/include
INCLUDE_DIRS := $(PYTHON_INCLUDE) /usr/local/include /usr/include/hdf5/
serial/
LIBRARY_DIRS := $(PYTHON_LIB) /usr/local/lib /usr/lib /usr/lib/arm-lin-
ux-gnueabihf/hdf5/serial/
```

Next, you can start to compile caffe

```
make all
make test
make runtest
make pycaffe
```

If you did't get any error above, congratulation on your success. Finally, please export pythonpath

```
sudo vim ~/.bashrc
export PYTHONPATH=/home/"$USER"/caffe/python:$PYTHONPATH
```

3) Step 3: try it out

Now, we can confirm whether the installation is successful. Download AlexNet and run caffe time

```
cd ~/caffe/
python scripts/download_model_binary.py models/bvlc_alexnet
./build/tools/caffe time -model models/bvlc_alexnet/deploy.prototxt
-weights  models/bvlc_alexnet/bvlc_alexnet.caffemodel   -iterations 10
```

And you can see the benchmark of AlexNet on Pi3 caffe.

2.2. Tensorflow

1) Step 1: install dependencies and clone repository

First, update apt-get:

```
$ sudo apt-get update
```

For Bazel:

```
$ sudo apt-get install pkg-config zip g++ zlib1g-dev unzip
```

For Tensorflow: (NCSDK only support python 3+. I didn't use mvNC on rpi3, so here I

choose python 2.7)
(For Python 2.7)

```
$ sudo apt-get install python-pip python-numpy swig python-dev
$ sudo pip install wheel
```

(For Python 3.3+)

```
$ sudo apt-get install python3-pip python3-numpy swig python3-dev
$ sudo pip3 install wheel
```

To be able to take advantage of certain optimization flags:

```
$ sudo apt-get install gcc-4.8 g++-4.8
$ sudo update-alternatives --install /usr/bin/gcc gcc /usr/bin/gcc-4.8
100
$ sudo update-alternatives --install /usr/bin/g++ g++ /usr/bin/g++-4.8
100
```

Make a directory that hold all the thing you need

```
$ mkdir tf
```

2) Step 2: build Bazel

Download and extract bazel (here I choose 0.7.0):

```
$ cd ~/tf
$ wget https://github.com/bazelbuild/bazel/releases/download/0.7.0/
bazel-0.7.0-dist.zip
$ unzip -d bazel bazel-0.7.0-dist.zip
```

Modify some file:

```
$ cd bazel
$ sudo chmod u+w ./* -R

$ nano scripts/bootstrap/compile.sh
```

To line 117, add “-J-Xmx500M”:

```
run "${JAVAC}" -classpath "${classpath}" -sourcepath "${sourcepath}" \
      -d "${output}/classes" -source "$JAVA_VERSION" -target "$JAVA_VERSION" \
      -encoding UTF-8 "@${paramfile}" -J-Xmx500M
```

Figure 2.1

```
$ nano tools/cpp/cc_configure.bzl
```

Place the line return “arm” around line 133 (beginning of the _get_cpu_value function):

```
...
"""Compute the cpu_value based on the OS name."""
return "arm"
...
```

Figure 2.2

Build Bazel (it will take a while, about 1 hour):

```
$ ./compile.sh
```

When the build finishes:

```
$ sudo cp output/bazel /usr/local/bin/bazel
```

Run bazel check if it’s working:

```
$ bazel
```

```
Usage: bazel <command> <options> ...

Available commands:
  analyze-profile     Analyzes build profile data.
  build               Builds the specified targets.
  canonicalize-flags  Canonicalizes a list of bazel options.
  clean               Removes output files and optionally stops the server.
  dump                Dumps the internal state of the bazel server process.
  fetch               Fetches external repositories that are prerequisites to the targets.
  help                Prints help for commands, or the index.
  info                Displays runtime info about the bazel server.
  mobile-install      Installs targets to mobile devices.
  query               Executes a dependency graph query.
  run                 Runs the specified target.
  shutdown            Stops the bazel server.
  test                Builds and runs the specified test targets.
  version             Prints version information for bazel.

Getting more help:
  bazel help <command>
                  Prints help and options for <command>.
  bazel help startup_options
                  Options for the JVM hosting bazel.
  bazel help target-syntax
                  Explains the syntax for specifying targets.
  bazel help info-keys
                  Displays a list of keys used by the info command.
```

Figure 2.3

3) Step 3: compile Tensorflow

Clone tensorflow repo (here I choose 1.4.0):

```
$ cd ~/tf
$ git clone -b r1.4 https://github.com/tensorflow/tensorflow.git
$ cd tensorflow
```

(Incredibly important) Changes references of 64-bit program implementations (which we don't have access to) to 32-bit implementations.

```
$ grep -Rl 'lib64' | xargs sed -i 's/lib64/lib/g'
```

Modify the file platform.h:

```
$ sed -i "s|#define IS_MOBILE_PLATFORM|//#define IS_MOBILE_PLATFORM|g"
tensorflow/core/platform/platform.h
```

Configure the build: (important) if you want to build for Python 3, specify /usr/bin/python3 for Python's location and /usr/local/lib/python3.x/dist-packages for the Python library path.

```
$ ./configure
Please specify the location of python. [Default is /usr/bin/python]: /usr/bin/python
Please specify optimization flags to use during compilation when bazel option "--config=opt"
Do you wish to use jemalloc as the malloc implementation? [Y/n] Y
Do you wish to build TensorFlow with Google Cloud Platform support? [y/N] N
Do you wish to build TensorFlow with Hadoop File System support? [y/N] N
Do you wish to build TensorFlow with the XLA just-in-time compiler (experimental)? [y/N] N
Please input the desired Python library path to use. Default is [/usr/local/lib/python2.7/di
Do you wish to build TensorFlow with OpenCL support? [y/N] N
Do you wish to build TensorFlow with CUDA support? [y/N] N
```

Figure 2.4

Build the Tensorflow (this will take a LOOOONG time, about 7 hrs):

```
$ bazel build -c opt --copt="-mfpu=neon-vfpv4" --copt="-funsafe-math-op-
timizations" --copt="-ftree-vectorize" --copt="-fomit-frame-pointer" --
local_resources 1024,1.0,1.0 --verbose_failures tensorflow/tools/
pip_package:build_pip_package
```

After finished compiling, install python wheel:

```
$ bazel-bin/tensorflow/tools/pip_package/build_pip_package /tmp/tensor-
flow_pkg
$ sudo pip install /tmp/tensorflow_pkg/tensorflow-1.4.0-cp27-none-lin-
ux_armv7l.whl
```

Check version:

```
$ python -c 'import tensorflow as tf; print(tf.__version__)'
```

And you're done! You deserve a break.

4) Step 3: try it out

Suppose you already have inception-v3 model (with inception-v3.meta and inception-v3.ckpt)

Create a testing python file

```
$ vim test.py
```

Write the following code:

```
1  import tensorflow as tf
2  import numpy as np
3  import cv2
4  import sys
5  import time
6
7  def run(input_image):
8      tf.reset_default_graph()
9      with tf.Session() as sess:
10         saver = tf.train.import_meta_graph('./output/inception-v3.meta')
11         saver.restore(sess, 'inception_v3.ckpt')
12
13         softmax_tensor = sess.graph.get_tensor_by_name('Softmax:0')
14         feed_dict = {'input:0': input_image}
15         classification = sess.run(softmax_tensor, {'input:0': input_image})    #first run fo warm-up
16
17         start_time = time.time()
18         classification = sess.run(softmax_tensor, {'input:0': input_image})
19         print 'predict label:', np.argmax(classification[0])
20         print 'predict time:', time.time() - start_time, 's'
21
22 if __name__=="__main__":
23     args = sys.argv
24     if len(args) != 2:
25         print 'Usage: python %s filename'%args[0]
26         quit()
27     image_data =tf.gfile.FastGFile(args[1], 'rb').read()
28     image = cv2.imread(args[1])
29     image = cv2.resize(image, (299,299))
30     image = np.array(image)/255.0
31     image = np.asarray(image).reshape((1, 299, 299, 3))
32     run(image)
```

Figure 2.5

Save, and excute it

```
$ python test.py cat.jpg
```

Then it will show the predict label and predict time.

UNIT A-3

Movidius Neural Compute Stick Install

3.1. Laptop Installation

install based on ncsdk website

```
git clone http://github.com/Movidius/ncsdk
cd ~/ncsdk
make install
make examples
```

test installation

```
cd ~/ncsdk/examples/app/hello_ncs_py/
make run
```

3.2. Duckiebot Installation

you only need to install the NCSDK but there is also the option of installing Caffe and/or Tensorflow as well, in order to perhaps speed up the development cycle. I would recommend against it, as it can be a bigger problem than it solves.

1) Barebones Install (recommended)

you don't need tensorflow, caffe, or any tools in order to run the compiled networks and not installing them will save you a lot of hassle

on duckiebot:

```
git clone http://github.com/Movidius/ncsdk
cd ~/ncsdk/api/src
make
sudo make install
```

2) Caffe/Tensorflow Install

> Note: if you want to be able to compile your models on the duckiebot itself, install tensorflow or caffe beforehand and remember to install for python 3 (`pip3`)

follow directions here

make sure caffe and tensorflow as installed

```
python3 -c 'import tensorflow as tf; import caffe'
```

install sdk:

```
git clone http://github.com/Movidius/ncsdk
cd ~/ncsdk
make install
make examples
```

UNIT A-4

How To Use Neural Compute Stick

4.1. Workflow

create and train model in tensorflow or caffe (brief note on configuration)

save tensorflow model as a `.meta` (or caffe model in `.prototxt`)

```
saver = tf.train.Saver()
...
saver.save(sess, 'model')
```

compile the model into NC format (documentation here)

```
mvNCCompile model.meta -o model.graph
```

move model onto duckiebot

```
scp model.meta user@robot name:~/path_to_networks/
```

run the compiled model

```
with open(path_to_networks + model.meta, mode='rb') as f:
    graphfile = f.read()
graph = device.AllocateGraph(graphfile)
graph.LoadTensor(input_image.astype(numpy.float16), 'user object')
output, userobj = graph.GetResult()
```

4.2. Benchmarking

get benchmarking (frames per second) from their app zoo

```
git clone https://github.com/movidius/ncappzoo
cd ncappzoo/apps/benchmarkncs
./mobilenets_benchmark.sh | grep FPSk
```

Part B

ETH Autonomous mobility on Demand 2019: Final Reports

Contents

UNIT B-1

Group name: final report

Before starting, you should have a look at some tips on how to write beautiful Duckiebook pages (unknown ref duckumentation/contribute)

warning next (1 of 4) index

warning

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

 > I do not know what is indicated by the link '#ducku-
mentation/contribute'.
```

Location not known more precisely.

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

.

The objective of this report is to bring justice to your extraordinarily hard work during the semester and make so that future generations of Duckietown students may take full advantage of it. Some of the sections of this report are repetitions from the preliminary design document (PDD) and intermediate report you have given.

1.1. The final result

Let's start from a teaser.

- Post a video of your best results (e.g., your demo video): remember to have duckies on the robots or something terrible might happen!

Add as a caption: see the operation manual to reproduce these results. Moreover, add a link to the readme.txt of your code.

1.2. Mission and Scope

Now tell your story:

Define what is your mission here.

1) Motivation

Now step back and tell us how you got to that mission.

- What are we talking about? [Brief introduction / problem in general terms]
- Why is it important? [Relevance]

2) Existing solution

- Was there a baseline implementation in Duckietown which you improved upon, or did you implemented from scratch? Describe the "prior work"

3) Opportunity

- What was wrong with the baseline / prior work / existing solution? Why did it need improvement?

Examples: - there wasn't a previous implementation - the previous performance, evaluated according to some specific metrics, was not satisfactory - it was not robust / reliable - somebody told me to do so (/s) (this is a terrible motivation. In general, never ever ever say "somebody told me to do it" or "everybody does like this")

- How did you go about improving the existing solution / approaching the problem? [contribution]

Examples: - We used method / algorithm xyz to fix the gap in knowledge (don't go in the details here) - Make sure to reference papers you used / took inspiration from, lessons, textbooks, third party projects and any other resource you took advantage of (check here how to add citations in this document). Even in your code, make sure you are giving credit in the comments to original authors if you are reusing some components.

4) Preliminaries

- Is there some particular theorem / "mathy" thing you require your readers to know before delving in the actual problem? Briefly explain it and links for more detailed explanations here.

Definition of link: - could be the reference to a paper / textbook - (bonus points) it is best if it is a link to Duckiebook chapter (in the dedicated "Preliminaries" section)

1.3. Definition of the problem

Up to now it was all fun and giggles. This is the most important part of your report: a crisp, possibly mathematical, definition of the problem you tackled. You can use part of the preliminary design document to fill this section.

Make sure you include your: - final objective / goal - assumptions made - quantitative performance metrics to judge the achievement of the goal

1.4. Contribution / Added functionality

Describe here, in technical detail, what you have done. Make sure you include: - a theoretical description of the algorithm(s) you implemented - logical architecture - software architecture - details on the actual implementation where relevant (how does the implementation differ from the theory?) - any infrastructure you had to develop in order to implement your algorithm - If you have collected a number of logs, add link to where you stored them

Feel free to create subsections when useful to ease the flow

1.5. Formal performance evaluation / Results

Be rigorous!

- For each of the tasks you defined in you problem formulation, provide quantitative results (i.e., the evaluation of the previously introduced performance metrics)
- Compare your results to the success targets. Explain successes or failures.
- Compare your results to the "state of the art" / previous implementation where relevant. Explain failure / success.
- Include an explanation / discussion of the results. Where things (as / better than / worst than) you expected? What were the biggest challenges?

1.6. Future avenues of development

Is there something you think still needs to be done or could be improved? List it here, and be specific!

Part C

ETH Autonomous mobility on Demand 2019: Final Reports

Contents

UNIT C-1

Demo template

Before starting, you should have a look at some tips on how to write beautiful Duckiebook pages (unknown ref duckumentation/contribute)

previous **warning** next (2 of 4) index

warning

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

 > I do not know what is indicated by the link '#ducku-
mentation/contribute'.
```

Location not known more precisely.

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

.

This is the template for the description of a demo. The spirit of this document is to be an operation manual, i.e., a straightforward, unambiguous recipe for reproducing the results of a specific behavior or set of behaviors.

It starts with the "knowledge box" that provides a crisp description of the border conditions needed:

- Duckiebot hardware configuration (see Duckiebot configurations)
- Duckietown hardware configuration (loops, intersections, robotarium, etc.)
- Number of Duckiebots
- Duckiebot setup steps

For example:

KNOWLEDGE AND ACTIVITY GRAPH

Requires: Duckiebot in configuration `DB18`

Requires: Duckietown without intersections

Requires: Camera calibration completed

1.1. Video of expected results

First, we show a video of the expected behavior (if the demo is successful).

Make sure the video is compliant with Duckietown, i.e. : the city meets the appearance specifications and the Duckiebots have duckies on board.

1.2. Duckietown setup notes

Here, describe the assumptions about the Duckietown, including:

- Layout (tiles types)

- Infrastructure (traffic lights, WiFi networks, ...) required
- Weather (lights, ...)

Do not write instructions on how to build the city here, unless you are doing something very particular that is not in the Duckietown operation manual (unknown ref opmanual_duckietown/duckietowns)

previous **warning** next (3 of 4) 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/duckietowns'.
```

Location not known more precisely.

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

. Here, merely point to them.

1.3. Duckiebot setup notes

Write here any special setup for the Duckiebot, if needed.

Do not repeat instructions here that are already included in the Duckiebot operation manual (unknown ref opmanual_duckiebot/opmanual_duckiebot)

previous **warning** (4 of 4) 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/opmanual_duckiebot'.
```

Location not known more precisely.

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

.

1.4. Pre-flight checklist

The pre-flight checklist describes the steps that are sufficient to ensure that the demo will be correct:

Check: operation 1 done

Check: operation 2 done

1.5. Demo instructions

Here, give step by step instructions to reproduce the demo.

Step 1: XXX

Step 2: XXX

Make sure you are specifying where to write each line of code that needs to be executed, and what should the expected outcome be. If there are typical pitfalls / errors you experienced, point to the next section for troubleshooting.

1.6. Troubleshooting

Add here any troubleshooting / tips and tricks required, in the form:

> **Symptom:** The Duckiebot flies
>
> **Resolution:** Unplug the battery and send an email to info@duckietown.org

> **Symptom:** I run `this elegant snippet of code` and get this error: `a nasty line of gibberish`
>
> **Resolution:** Power cycle until it works.

1.7. Demo failure demonstration

Finally, put here video of how the demo can fail, when the assumptions are not respected.

You can upload the videos to the Duckietown Vimeo account and link them here.

Other learning modules

Logo