**Group Project**

**JetBot Teleoperation & Lane Following**

**The A-Team**

Bryce Leeper, Jared Jess, Michael Surbeck

12/9/22

MRE-320 Sensors and Actuators

Fall ‘22

**Strategy**

Our strategy to win the manually controlled race was pretty straight forward. We used a game controller to control the JetBot for the teleoperate race. The idea was to be able to accurately turn the JetBot while moving and accelerating quickly. Our strategy for the automatic race included taking pictures so the JetBot could detect the lines of the track, specifically the solid black line. This would allow the JetBot to focus more on the middle of the track, allowing it to smoothly move down the straights at speed and giving it wiggle room during the turns. The JetBot stores these pictures and uses them to “train” itself, allowing for a smooth drive around the track..

**Implementation**

We used an Xbox 360 controller to input commands to the JetBot. The JetBot was controlled with the controller’s joysticks, where the left joystick controlled the speed of the left wheel and the right joystick controlled the speed of the right wheel. Pushing forward caused the wheels to move forward and pushing backwards caused the wheels to move backwards. We did this because we believed it gave us the finest control over the JetBot while not suffering any loss in power. For lane-following operations, the JetBot needed data to use while navigating. We used the on-board camera to take pictures of the track, added a point to the picture (where you want the JetBot to travel to), then saved the picture to a folder. We used a dataset of 300 pictures which included pictures of ideal situations, ex. in the middle of the track facing forward, and sub-par positions, ex. one wheel outside of the track facing 35º off center. Doing this ensured that the robot would know where to go when going off the track. To tune the JetBot for automated driving around the track, we first had the JetBot drive around with the tuning sliders turned all the way down, except the speed gain otherwise it wouldn’t move, and the steering bias was set to 0. We then proceeded to tweak the steering gain, steering kd, and steering bias after a few laps. The process was repeated until we were satisfied that our JetBot would consistently move around the track with little to no errors at a smooth and quick pace.

**Results**

Our results were very satisfactory when compared to the team’s expectations. We had a very quick lap time for the teleoperated section that only suffered from one or two issues. We could not figure out how to tune the controller like we wanted but it was still manageable. Our automated operations worked perfectly during the testing phase, even completing perfect laps while moving in the opposite direction from our data retrieval. However, we had a few issues with the JetBot getting lost on the track during the presentation. Our bot still completed its three laps very quickly.

**Changes**

Our team would not make very many changes to our JetBot if given the chance. For the teleoperated portion, we would probably change the control scheme. Instead of each joystick controlling power to a respective wheel, we believe that it would be easier to control if the left joystick controlled the speed forwards and backwards while the right joystick could be used for steering. For the automated portion we wouldn’t change much, if any. Although we fell off the track, all sessions before showed no signs of error. We could have added more pictures to help avoid the mistake made during presentation, but we think the WIFI may have a hand in the issue.

**Appendix**

Pictures

A screenshot of a computer

Description automatically generated with medium confidence

Figure1: Control Sliders with Parameters Set

A screenshot of a computer

Description automatically generated with low confidence

Figure2: Sliders Used to See What the JetBot was “Thinking”

Manual Operations Code

from jetbot import Robot #import robot class

robot = Robot() #initialize class

#initalize controller to be used

import ipywidgets.widgets as widgets

controller = widgets.Controller(index=0)

display(controller)

#linking the controller to JetBot controls

from jetbot import Robot

import traitlets

robot = Robot()

left\_link=traitlets.dlink((controller.button[6],'value'),(robot.left\_motor,'value'),transform=lambda x:-x)

right\_link=traitlets.dlink((controller.button[7],'value'),(robot.right\_motor,'value'),transform=lambda x:-x)

Road Following Code

import torch

device = torch.device('cuda')

# load the trained data to JetBot

import torch

from torch2trt import TRTModule

model\_trt = TRTModule()

model\_trt.load\_state\_dict(torch.load('best\_steering\_model\_xy\_trt.pth'))

# creating the pre-processing function to format of pictures

import torchvision.transforms as transforms

import torch.nn.functional as F

import cv2

import PIL.Image

import numpy as np

mean = torch.Tensor([0.485, 0.456, 0.406]).cuda().half()

std = torch.Tensor([0.229, 0.224, 0.225]).cuda().half()

def preprocess(image):

image = PIL.Image.fromarray(image)

image = transforms.functional.to\_tensor(image).to(device).half()

image.sub\_(mean[:, None, None]).div\_(std[:, None, None])

return image[None, ...]

# start and link camera

from IPython.display import display

import ipywidgets

import traitlets

from jetbot import Camera, bgr8\_to\_jpeg

camera = Camera()

from jetbot import Robot

robot = Robot()

#slider control with our tune parameters hard coded

speed\_gain\_slider = ipywidgets.FloatSlider(min=0.0, max=1.0, step=0.01, value=0.29, description='speed gain')

steering\_gain\_slider = ipywidgets.FloatSlider(min=0.0, max=1.0, step=0.01, value=0.08, description='steering gain')

steering\_dgain\_slider = ipywidgets.FloatSlider(min=0.0, max=0.5, step=0.001, value=0.5, description='steering kd')

steering\_bias\_slider = ipywidgets.FloatSlider(min=-0.3, max=0.3, step=0.01, value=0.0, description='steering bias')

display(speed\_gain\_slider, steering\_gain\_slider, steering\_dgain\_slider, steering\_bias\_slider)

# real time sliders

x\_slider = ipywidgets.FloatSlider(min=-1.0, max=1.0, description='x')

y\_slider = ipywidgets.FloatSlider(min=0, max=1.0, orientation='vertical', description='y')

steering\_slider = ipywidgets.FloatSlider(min=-1.0, max=1.0, description='steering')

speed\_slider = ipywidgets.FloatSlider(min=0, max=1.0, orientation='vertical', description='speed')

display(ipywidgets.HBox([y\_slider, speed\_slider]))

display(x\_slider, steering\_slider)

# camera function for operations

angle = 0.0

angle\_last = 0.0

def execute(change):

global angle, angle\_last

image = change['new']

xy = model\_trt(preprocess(image)).detach().float().cpu().numpy().flatten()

x = xy[0]

y = (0.5 - xy[1]) / 2.0

x\_slider.value = x

y\_slider.value = y

speed\_slider.value = speed\_gain\_slider.value

angle = np.arctan2(x, y)

pid = angle \* steering\_gain\_slider.value + (angle - angle\_last) \* steering\_dgain\_slider.value

angle\_last = angle

steering\_slider.value = pid + steering\_bias\_slider.value

robot.left\_motor.value = max(min(speed\_slider.value + steering\_slider.value, 1.0), 0.0)

robot.right\_motor.value = max(min(speed\_slider.value - steering\_slider.value, 1.0), 0.0)

execute({'new': camera.value})

# attach camera to camera function

amera.observe(execute, names='value')