ECE140A Lab 2 - Part 2: Road Following

Due: 27 January 2020 (Monday Lab session)
29 January 2020 (Wednesday Lab session)
31 January 2020 (Friday Lab session)

Road Following

If you've run through the collision avoidance sample, your should be familiar following three steps Deployment

1. Data collection
2. Training
3. Deployment

In this notebook, we'll do the same exact thing! Except, instead of classification, you'll learn a different fundamental technique, **regression**, that we'll use to enable JetBot to follow a road (or really, any path or target point).

1. Place the JetBot in different positions on a path (offset from center, different angles, etc)

> Remember from collision avoidance, data variation is key!

1. Display the live camera feed from the robot
2. Using a gamepad controller, place a 'green dot', which corresponds to the target direction we want the robot to travel, on the image.
3. Store the X, Y values of this green dot along with the image from the robot's camera

Then, in the training notebook, we'll train a neural network to predict the X, Y values of our label. In the live demo, we'll use the predicted X, Y values to compute an approximate steering value (it's not 'exactly' an angle, as that would require image calibration, but it's roughly proportional to the angle so our controller will work fine).

So how do you decide exactly where to place the target for this example? Here is a guide we think may help

1. Look at the live video feed from the camera
2. Imagine the path that the robot should follow (try to approximate the distance it needs to avoid running off road etc.)
3. Place the target as far along this path as it can go so that the robot could head straight to the target without 'running off' the road.

> For example, if we're on a very straight road, we could place it at the horizon. If we're on a sharp turn, it may need to be placed closer to the robot so it doesn't run out of boundaries.

Assuming our deep learning model works as intended, these labeling guidelines should ensure the following:

1. The robot can safely travel directly towards the target (without going out of bounds etc.)
2. The target will continuously progress along our imagined path

What we get, is a 'carrot on a stick' that moves along our desired trajectory. Deep learning decides where to place the carrot, and JetBot just follows it :)

Labeling example video

Execute the block of code to see an example of how to we labeled the images. This model worked after only 123 images :)

In [2]:
```
from IPython.display import HTML
HTML('<iframe width="560" height="315" src="https://www.youtube.com/embed/FW4En6LejhI" frameborder="0" allow="accelerometer; autoplay; encrypted-media; gyroscope; picture-in-picture" allowfullscreen></iframe>')
```

Out[2]:

Import Libraries

So lets get started by importing all the required libraries for "data collection" purpose. We will mainly use OpenCV to visualize and save image with labels. Libraries such as uuid, datetime are used for image naming.

In [3]:
```
# IPython Libraries for display and widgets
import traitlets
import ipywidgets.widgets as widgets
from IPython.display import display

# Camera and Motor Interface for JetBot
from jetbot import Robot, Camera, bgr8_to_jpeg

# Python basic packages for image annotation
from uuid import uuid1
import os
import json
import glob
import datetime
import numpy as np
import cv2
import time
```

Display Live Camera Feed

First, let's initialize and display our camera like we did in the teleoperation notebook.

We use Camera Class from JetBot to enable CSI MIPI camera. Our neural network takes a 224x224 pixel image as input. We'll set our camera to that size to minimize the filesize of our dataset (we've tested that it works for this task). In some scenarios it may be better to collect data in a larger image size and downscale to the desired size later.

In [4]:
```
camera = Camera()

image_widget = widgets.Image(format='jpeg', width=224, height=224)
target_widget = widgets.Image(format='jpeg', width=224, height=224)

x_slider = widgets.FloatSlider(min=-1.0, max=1.0, step=0.001, description='x')
y_slider = widgets.FloatSlider(min=-1.0, max=1.0, step=0.001, description='y')

def display_xy(camera_image):
    image = np.copy(camera_image)
    x = x_slider.value
    y = y_slider.value
    x = int(x * 224 / 2 + 112)
    y = int(y * 224 / 2 + 112)
    image = cv2.circle(image, (x, y), 8, (0, 255, 0), 3)
    image = cv2.circle(image, (112, 224), 8, (0, 0,255), 3)
    image = cv2.line(image, (x,y), (112,224), (255,0,0), 3)
    jpeg_image = bgr8_to_jpeg(image)
    return jpeg_image

time.sleep(1)
traitlets.dlink((camera, 'value'), (image_widget, 'value'), transform=bgr8_to_jpe
g)
traitlets.dlink((camera, 'value'), (target_widget, 'value'), transform=display_xy)

display(widgets.HBox([image_widget, target_widget]), x_slider, y_slider)
```

Create Gamepad Controller

This step is similar to "Teleoperation" task. In this task, we will use gamepad controller to label images.

The first thing we want to do is create an instance of the Controller widget, which we'll use to label images with "x" and "y" values as mentioned in introduction. The Controller widget takes a index parameter, which specifies the number of the controller. This is useful in case you have multiple controllers attached, or some gamepads appear as multiple controllers. To determine the index of the controller you're using,

Visit http://html5gamepad.com (http://html5gamepad.com). Press buttons on the gamepad you're using Remember the index of the gamepad that is responding to the button presses Next, we'll create and display our controller using that index.

In [5]:
```
controller = widgets.Controller(index=0)

display(controller)
```

Connect Gamepad Controller to Label Images

Now, even though we've connected our gamepad, we haven't yet attached the controller to label images! We'll connect that to the left and right vertical axes using the dlink function. The dlink function, unlike the link function, allows us to attach a transform between the source and target.

In [6]:
```
widgets.jsdlink((controller.axes[2], 'value'), (x_slider, 'value'))
widgets.jsdlink((controller.axes[3], 'value'), (y_slider, 'value'))
```

Collect data

The following block of code will display the live image feed, as well as the number of images we've saved. We store the target X, Y values by

1. Place the green dot on the target
2. Press 'down' on the DPAD to save

This will store a file in the `dataset_xy` folder with files named

`xy_<x value>_<y value>_<uuid>.jpg`

When we train, we load the images and parse the x, y values from the filename

In [7]:
```
DATASET_DIR = 'dataset_xy'

# we have this "try/except" statement because these next functions can throw an er
ror if the directories exist already
try:
    os.makedirs(DATASET_DIR)
except FileExistsError:
    print('Directories not created becasue they already exist')

for b in controller.buttons:
    b.unobserve_all()

count_widget = widgets.IntText(description='count', value=len(glob.glob(os.path.jo
in(DATASET_DIR, '*.jpg'))))

def xy_uuid(x, y):
    return 'xy_%03d_%03d_%s' % (x * 50 + 50, y * 50 + 50, uuid1())

def save_snapshot(change):
    if change['new']:
        uuid = xy_uuid(x_slider.value, y_slider.value)
        image_path = os.path.join(DATASET_DIR, uuid + '.jpg')
        with open(image_path, 'wb') as f:
            f.write(image_widget.value)
        count_widget.value = len(glob.glob(os.path.join(DATASET_DIR, '*.jpg')))

#buttons[13] -> buttons[7]: start button
controller.buttons[7].observe(save_snapshot, names='value')

display(widgets.VBox([
    target_widget,
    count_widget
]))
```

```
Directories not created becasue they already exist
```

Next

Once you've collected enough data, we'll need to copy that data to our GPU desktop or cloud machine for training. First, we can call the following terminal command to compress our dataset folder into a single zip file.

> If you're training on the JetBot itself, you can skip this step!

The ! prefix indicates that we want to run the cell as a shell (or terminal) command.

The -r flag in the zip command below indicates recursive so that we include all nested files, the -q flag indicates quiet so that the zip command doesn't print any output

In [2]:
```
def timestr():
    return str(datetime.datetime.now().strftime('%Y-%m-%d_%H-%M-%S'))

!zip -r -q road_following_{DATASET_DIR}_{timestr()}.zip {DATASET_DIR}
```

```
/bin/sh: 1: Syntax error: "(" unexpected
```

You should see a file named road*following*<Date&Time>.zip in the Jupyter Lab file browser. You should download the zip file using the Jupyter Lab file browser by right clicking and selecting Download.

Road Follower - Train Model

In this notebook we will train a neural network to take an input image, and output a set of x, y values corresponding to a target.

We will be using PyTorch deep learning framework to train ResNet18 neural network architecture model for road follower application.

In [3]:
```
import torch
import torch.optim as optim
import torch.nn.functional as F
import torchvision
import torchvision.datasets as datasets
import torchvision.models as models
import torchvision.transforms as transforms
import glob
import PIL.Image
import os
import numpy as np
```

Download and extract data

Before you start, you should upload the `road_following_<Date&Time>.zip` file that you created in the `data_collection.ipynb` notebook on the robot.

> If you're training on the JetBot you collected data on, you can skip this!

You should then extract this dataset by calling the command below:

In [5]:
```
!unzip -q road_following_dataset_xy.zip
```

```
unzip:  cannot find or open road_following_dataset_xy.zip, road_following_datase
t_xy.zip.zip or road_following_dataset_xy.zip.ZIP.
```

You should see a folder named `dataset_all` appear in the file browser.

Create Dataset Instance

Here we create a custom `torch.utils.data.Dataset` implementation, which implements the `__len__` and `__getitem__` functions. This class is responsible for loading images and parsing the x, y values from the image filenames. Because we implement the `torch.utils.data.Dataset` class, we can use all of the torch data utilities :)

We hard coded some transformations (like color jitter) into our dataset. We made random horizontal flips optional (in case you want to follow a non-symmetric path, like a road where we need to 'stay right'). If it doesn't matter whether your robot follows some convention, you could enable flips to augment the dataset.

In [6]:
```
def get_x(path):
    """Gets the x value from the image filename"""
    return (float(int(path[3:6])) - 50.0) / 50.0

def get_y(path):
    """Gets the y value from the image filename"""
    return (float(int(path[7:10])) - 50.0) / 50.0

class XYDataset(torch.utils.data.Dataset):

    def __init__(self, directory, random_hflips=False):
        self.directory = directory
        self.random_hflips = random_hflips
        self.image_paths = glob.glob(os.path.join(self.directory, '*.jpg'))
        self.color_jitter = transforms.ColorJitter(0.3, 0.3, 0.3, 0.3)

    def __len__(self):
        return len(self.image_paths)

    def __getitem__(self, idx):
        image_path = self.image_paths[idx]

        image = PIL.Image.open(image_path)
        x = float(get_x(os.path.basename(image_path)))
        y = float(get_y(os.path.basename(image_path)))

        if float(np.random.rand(1)) > 0.5:
            image = transforms.functional.hflip(image)
            x = -x

        image = self.color_jitter(image)
        image = transforms.functional.resize(image, (224, 224))
        image = transforms.functional.to_tensor(image)
        image = image.numpy()[::-1].copy()
        image = torch.from_numpy(image)
        image = transforms.functional.normalize(image, [0.485, 0.456, 0.406], [0.2
29, 0.224, 0.225])

        return image, torch.tensor([x, y]).float()

dataset = XYDataset('dataset_xy', random_hflips=False)
```

Split dataset into train and test sets

Once we read dataset, we will split data set in train and test sets. In this example we split train and test a 90%-10%. The test set will be used to verify the accuracy of the model we train.

In [12]:
```
test_percent = 0.1
num_test = int(test_percent * len(dataset))
train_dataset, test_dataset = torch.utils.data.random_split(dataset, [len(dataset)
- num_test, num_test])
```

Create data loaders to load data in batches

We use `DataLoader` class to load data in batches, shuffle data and allow using multi-subprocesses. In this example we use batch size of 64. Batch size will be based on memory available with your GPU and it can impact accuracy of the model.

In [13]:
```
train_loader = torch.utils.data.DataLoader(
    train_dataset,
    batch_size=4,
    shuffle=True,
    num_workers=2
)

test_loader = torch.utils.data.DataLoader(
    test_dataset,
    batch_size=4,
    shuffle=True,
    num_workers=2
)
```

Define Neural Network Model

We use ResNet-18 model available on PyTorch TorchVision.

In a process called transfer learning, we can repurpose a pre-trained model (trained on millions of images) for a new task that has possibly much less data available.

More details on ResNet-18 : https://github.com/pytorch/vision/blob/master/torchvision/models/resnet.py (https://github.com/pytorch/vision/blob/master/torchvision/models/resnet.py)

More Details on Transfer Learning: https://www.youtube.com/watch?v=yofjFQddwHE (https://www.youtube.com/watch?v=yofjFQddwHE)

In [14]:
```
model = models.resnet18(pretrained=True)
```

ResNet model has fully connect (fc) final layer with 512 as `in_features` and we will be training for regression thus `out_features` as 1

Finally, we transfer our model for execution on the GPU

In [15]:
```
model.fc = torch.nn.Linear(512, 2)
device = torch.device('cuda')
model = model.to(device)
```

Train Regression:

We train for 50 epochs and save best model if the loss is reduced.

In [16]:
```
NUM_EPOCHS = 70
BEST_MODEL_PATH = 'best_steering_model_xy.pth'
best_loss = 1e9

optimizer = optim.Adam(model.parameters())

for epoch in range(NUM_EPOCHS):

    model.train()
    train_loss = 0.0
    for images, labels in iter(train_loader):
        images = images.to(device)
        labels = labels.to(device)
        optimizer.zero_grad()
        outputs = model(images)
        loss = F.mse_loss(outputs, labels)
        train_loss += loss
        loss.backward()
        optimizer.step()
    train_loss /= len(train_loader)

    model.eval()
    test_loss = 0.0
    for images, labels in iter(test_loader):
        images = images.to(device)
        labels = labels.to(device)
        outputs = model(images)
        loss = F.mse_loss(outputs, labels)
        test_loss += loss
    test_loss /= len(test_loader)

    print('%f, %f' % (train_loss, test_loss))
    if test_loss < best_loss:
        torch.save(model.state_dict(), BEST_MODEL_PATH)
        best_loss = test_loss
```

```
0.692535, 0.411308
0.160455, 0.176489
0.144431, 0.276675
0.148823, 0.191925
0.144605, 0.088278
0.133040, 0.203298
0.119821, 0.139332
0.119693, 0.148735
0.103862, 0.145626
0.123379, 0.146391
0.120463, 0.127433
0.115348, 0.222227
0.102176, 0.144351
0.111646, 0.099933
0.085273, 0.125446
0.092366, 0.176615
0.077745, 0.127874
0.097331, 0.122127
0.085938, 0.194923
0.095425, 0.152073
0.117951, 0.134613
0.088901, 0.104124
0.092443, 0.112411
0.093398, 0.125461
0.078201, 0.135913
0.075255, 0.184990
0.100428, 0.178787
0.081771, 0.100254
0.087981, 0.174237
0.087182, 0.201433
0.090174, 0.123504
0.076400, 0.185916
0.072255, 0.119152
0.074607, 0.111783
0.066404, 0.132506
0.076354, 0.156779
0.058767, 0.068307
0.071977, 0.117091
0.077918, 0.194971
0.081635, 0.107820
0.070314, 0.113088
0.079968, 0.107012
0.076286, 0.151532
0.087250, 0.119517
0.071749, 0.128412
0.071963, 0.167700
0.074774, 0.124316
0.059600, 0.137237
0.075920, 0.162360
0.071224, 0.136863
0.075672, 0.124763
0.068970, 0.136375
0.082377, 0.146068
0.057118, 0.131255
0.067124, 0.138703
0.073707, 0.117266
0.067135, 0.104715
0.074151, 0.156406
0.077076, 0.165144
0.067438, 0.191842
0.068334, 0.113565
0.073478, 0.121358
0.065243, 0.157554
0.067561, 0.114635
0.066507, 0.164533
```

Once the model is trained, it will generate `best_steering_model_xy.pth` file which you can use for inferencing in the live demo notebook.

If you trained on a different machine other than JetBot, you'll need to upload this to the JetBot to the `road_following` example folder.

Once the model is trained, it will generate best_steering_model_xy.pth file which you can use for inferencing in the live demo notebook.

If you trained on a different machine other than JetBot, you'll need to upload this to the JetBot to the road_following example folder.

Road Following - Live demo

In this notebook, we will use model we trained to move jetBot smoothly on track.

Load Trained Model

We will assume that you have already downloaded `best_steering_model_xy.pth` to work station as instructed in "train_model.ipynb" notebook. Now, you should upload model file to JetBot in to this notebooks's directory. Once that's finished there should be a file named `best_steering_model_xy.pth` in this notebook's directory.

> Please make sure the file has uploaded fully before calling the next cell

Execute the code below to initialize the PyTorch model. This should look very familiar from the training notebook.

In [7]:
```
import torchvision
import torch

model = torchvision.models.resnet18(pretrained=False)
model.fc = torch.nn.Linear(512, 2)
```

Next, load the trained weights from the `best_steering_model_xy.pth` file that you uploaded.

In [8]:
```
model.load_state_dict(torch.load('best_steering_model_xy.pth'))
```

Currently, the model weights are located on the CPU memory execute the code below to transfer to the GPU device.

In [9]:
```
device = torch.device('cuda')
model = model.to(device)
model = model.eval().half()
```

Creating the Pre-Processing Function

We have now loaded our model, but there's a slight issue. The format that we trained our model doesnt exactly match the format of the camera. To do that, we need to do some preprocessing. This involves the following steps:

1. Convert from HWC layout to CHW layout
2. Normalize using same parameters as we did during training (our camera provides values in [0, 255] range and training loaded images in [0, 1] range so we need to scale by 255.0
3. Transfer the data from CPU memory to GPU memory
4. Add a batch dimension

In [10]:
```
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, ...]
```

Awesome! We've now defined our pre-processing function which can convert images from the camera format to the neural network input format.

Now, let's start and display our camera. You should be pretty familiar with this by now.

In [11]:
```
from IPython.display import display
import ipywidgets
import traitlets
from jetbot import Camera, bgr8_to_jpeg

camera = Camera()

image_widget = ipywidgets.Image()

traitlets.dlink((camera, 'value'), (image_widget, 'value'), transform=bgr8_to_jpe
g)

display(image_widget)
```

We'll also create our robot instance which we'll need to drive the motors.

In [12]:
```
from jetbot import Robot

robot = Robot()
```

Now, we will define sliders to control JetBot

> Note: We have initialize the slider values for best known configurations, however these might not work for your dataset, therefore please increase or decrease the sliders according to your setup and environment

1. Speed Control (speed_gain_slider): To start your JetBot increase `speed_gain_slider`
2. Steering Gain Control (steering_gain_sloder): If you see JetBot is woblling, you need to reduce `steering_gain_slider` till it is smooth
3. Steering Bias control (steering_bias_slider): If you see JetBot is biased towards extreme right or extreme left side of the track, you should control this slider till JetBot start following line or track in the center. This accounts for motor biases as well as camera offsets

> Note: You should play around above mentioned sliders with lower speed to get smooth JetBot road following behavior.

In [13]:
```
speed_gain_slider = ipywidgets.FloatSlider(min=0.0, max=1.0, step=0.01, descriptio
n='speed gain')
steering_gain_slider = ipywidgets.FloatSlider(min=0.0, max=1.0, step=0.01, value=
0.2, description='steering gain')
steering_dgain_slider = ipywidgets.FloatSlider(min=0.0, max=0.5, step=0.001, value
=0.0, 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_b
ias_slider)
```

Next, let's display some sliders that will let us see what JetBot is thinking. The x and y sliders will display the predicted x, y values.

The steering slider will display our estimated steering value. Please remember, this value isn't the actual angle of the target, but simply a value that is nearly proportional. When the actual angle is `0`, this will be zero, and it will increase / decrease with the actual angle.

In [14]:
```
x_slider = ipywidgets.FloatSlider(min=-1.0, max=1.0, description='x')
y_slider = ipywidgets.FloatSlider(min=0, max=1.0, orientation='vertical', descript
ion='y')
steering_slider = ipywidgets.FloatSlider(min=-1.0, max=1.0, description='steering
')
speed_slider = ipywidgets.FloatSlider(min=0, max=1.0, orientation='vertical', desc
ription='speed')

display(ipywidgets.HBox([y_slider, speed_slider]))
display(x_slider, steering_slider)
```

Next, we'll create a function that will get called whenever the camera's value changes. This function will do the following steps

1. Pre-process the camera image
2. Execute the neural network
3. Compute the approximate steering value
4. Control the motors using proportional / derivative control (PD)

In [15]:
```
angle = 0.0
angle_last = 0.0

def execute(change):
    global angle, angle_last
    image = change['new']
    xy = model(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_dga
in_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})
```

Cool! We've created our neural network execution function, but now we need to attach it to the camera for processing.

We accomplish that with the observe function.

> WARNING: This code will move the robot!! Please make sure your robot has clearance and it is on Lego or Track you have collected data on. The road follower should work, but the neural network is only as good as the data it's trained on!

In [29]:
```
camera.observe(execute, names='value')
```

Awesome! If your robot is plugged in it should now be generating new commands with each new camera frame.

You can now place JetBot on Lego or Track you have collected data on and see whether it can follow track.

If you want to stop this behavior, you can unattach this callback by executing the code below.

In [30]:
```
camera.unobserve(execute, names='value')
robot.stop()
```

Conclusion

That's it for this live demo! Hopefully you had some fun seeing your JetBot moving smoothly on track following the road!!!

If your JetBot wasn't following road very well, try to spot where it fails. The beauty is that we can collect more data for these failure scenarios and the JetBot should get even better :)

In [31]:
```
robot.stop()
```

https://www.youtube.com/watch?v=VqWWwaKR_DE&feature=youtu.be (https://www.youtube.com/watch?v=VqWWwaKR_DE&feature=youtu.be)

In [ ]:
```

```