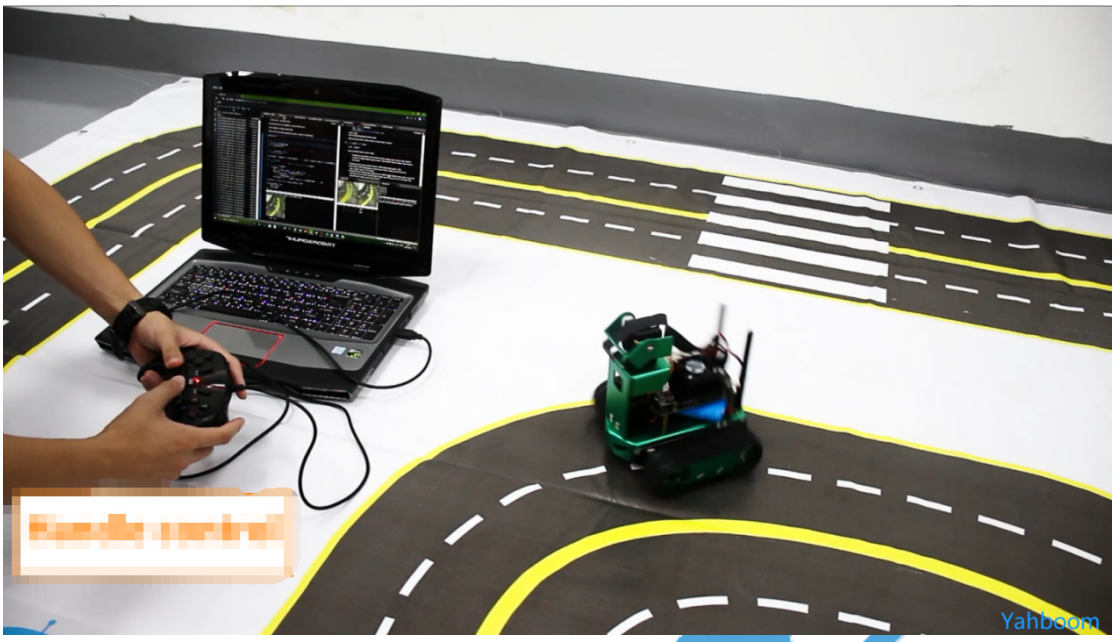
5.6 Autopilot

1. Collect data through Jetbot



If you have already browsed the collision avoidance example, you should be familiar with the following three steps.

1. Data collection
2. Training
3. Deployment

In this case, we will do the same thing. However, in addition to classification, you will learn another basic technique, 'regression'-regression, which we will use to enable Jetbot to follow a path (actually any path or target point).

1. Place the Jetbot at different locations on the path (offset from the center, different angles, etc.).
2. Display the live camera input from Robot 3.
3. Using the gamepad controller, place a “green dot” on the image that corresponds to the target direction we want the robot to move.
4. Store the X, Y values of this green dot along with the image of the robot camera.

Then, in the training notebook, we will train a neural network to predict the X, Y values of our labels. In the live demo, we will use the predicted X, Y values to calculate an approximate steering value (it is not an "exact" angle, because this requires image calibration, but it is roughly proportional to the angle, so our controller Will work normally).

So, how to determine the target position of this example?

Here are some guidelines that we think might be helpful:

1. Look at the live video of the camera.
2. Imagine the path that the robot should follow (try to get close to the distance it needs to avoid running off the road, etc.).

3. Place the target as far as possible so that the robot can rush directly to the target without “running away” from the road.

For example, if we are on a very straight road, we can put it on the horizon. If we are making a sharp turn, it may need to be placed closer to the robot so that it does not run out of the border.

Assuming our deep learning model works as expected, these markup guidelines should ensure the following:
1. The robot can move directly to the target safely (not out of bounds, etc.)
2. The goal will continue to move along the path we imagined
What we got was a carrot on the big road. The carrot moved along the trajectory we wanted. Deep learning decided where to put the carrot, and Jetbot just followed it.
We start by importing all the libraries needed for "data collection." We will primarily use OpenCV to visualize and save images with labels.
Uuid, datetime and other libraries for image naming:

```
# IPython libraries be used to diplay and parts
import traitlets
import ipywidgets.widgets as widgets
from IPython.display import display
# Camera and servo interface foe Jetbot robot car
from jetbot import Robot, Camera, bgr8_to_jpeg
# Python base package for image annotations
from uuid import uuid1
import os
import json
import glob
import datetime
import numpy as np
import cv2
import time
from servoserial import ServoSerial
import threading
# Kill pthread
import inspect
import ctypes
```

Yahboom

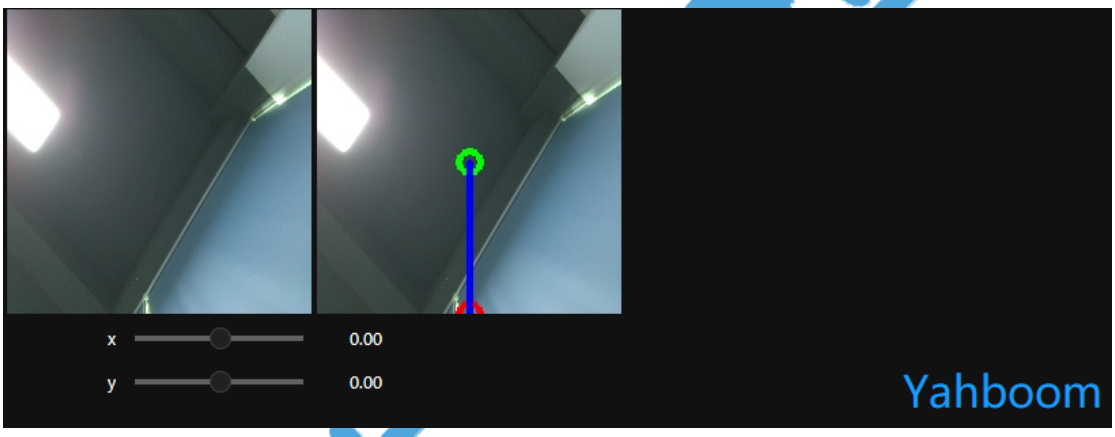
Our neural network takes an image of 224x224 pixels as input. We set the camera to this parameter to minimize the file size of the dataset (we have tested it for this task).

In some scenarios, it's best to collect the data at a larger image size and then reduce it to the desired size.

```
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_jpeg)
traitlets.dlink((camera, 'value'), (target_widget, 'value'), transform=display_xy)

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

Yahboom

After running the above code, the following interface will be displayed below the upper cell:



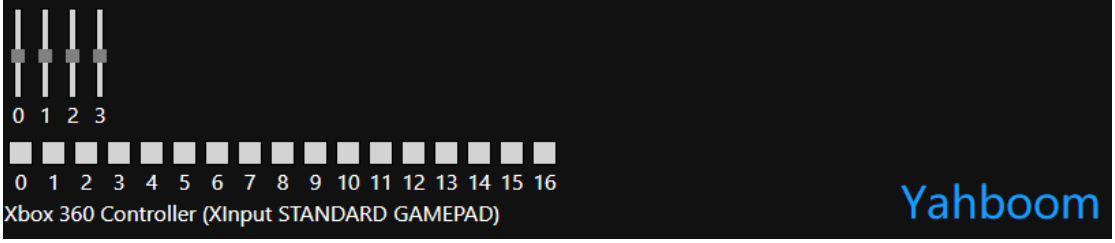
This step is similar to the "handle remote" task. In this example, we will use the gamepad controller to mark the image.

First, we have to do is create an instance of the Controller widget, which we will use to mark the image with "x" and "y" values, as described in the introduction. The Controller widget accepts an index parameter that specifies the number of controllers. This is useful if you have multiple controllers, or if some gamepads appear as multiple controllers. To determine the index of the controller we are using, then before we create the handle instance we will follow the steps we have just learned to use the remote control handle:

1. Visit http://html5gamepad.com
2. Press the button on the gamepad you are using
3. Remember the index of the gamepad that responds to the button

Then, we will use this index to create and display the controller.

```
controller = widgets.Controller(index=0)
robot = Robot()
display(controller)
```



Next, we connect the Gamepad controller to the label image.

Connect the Gamepad controller to the label image.

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

Yahboom

The code below will display the live image feed and the number of images we saved.
We store the value of the target X, Y:
1. Put the green dot on the target
2. Press the 13th button to save
Then the data we want will be saved to the ``dataset_xy`` folder. The saved file naming format is:
``xy_<x value>_<y value>_<uuid>.jpg``
When we train, we load the image and parse the x and y values in the file name.
Code shown below:

```
DATASET_DIR = 'dataset_xy'

# we have this "try/except" statement because these next functions can throw an error if the directories exist al
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.join(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')))

controller.buttons[13].observe(save_snapshot, names='value')

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

Yahboom

For ease to using, we can open a new thread to control the Jetbot robot car by handled to collect data through the handle:

Start pthread

```
thread1 = threading.Thread(target=jetbot_motion)
thread1.setDaemon(True)
thread1.start()
```

Add method of close pthread

```
def _async_raise(tid, exctype):
    """raises the exception, performs cleanup if needed"""
    tid = ctypes.c_long(tid)
    if not inspect.isclass(exctype):
        exctype = type(exctype)
    res = ctypes.pythonapi.PyThreadState_SetAsyncExc(tid, ctypes.py_object(exctype))
    if res == 0:
        raise ValueError("invalid thread id")
    elif res != 1:
        # """if it returns a number greater than one, you're in trouble,
        # and you should call it again with exc=NULL to revert the effect"""
        ctypes.pythonapi.PyThreadState_SetAsyncExc(tid, None)


def stop_thread(thread):
    _async_raise(thread.ident, SystemExit)
```

Create a method to adjust the position of the Jetbot to the autopilot angle and call this method to adjust.

```
servo_device = ServoSerial()
def camservoInitFunction():
    global leftrightpulse, updownpulse
    leftrightpulse = 2048
    updownpulse = 2048
    servo_device.Servo_serial_control(1, 2048)
    time.sleep(0.1)
    servo_device.Servo_serial_control(2, 1300)
```

Yahboom

Call the above method to adjust the pan/tilt to the autopilot angle

```
camservoInitFunction()
```

Collect data as much as possible in accordance with the method I mentioned above, otherwise it may lead to inaccurate recognition when driving automatically.

The corresponding complete source code is located:
/home/jetbot/Notebook/17.Autopilot-Basic/Data collection.ipynb

2. Train the neural network model

We will train a neural network to get an input image and output a set of x, y values corresponding to a target.

We will use the PyTorch deep learning framework we used in the previous course to train the ResNet18 neural network structure model to identify road conditions for automatic driving.
First, we need to import all the required data packets:

```
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
```

Yahboom

We create a custom 'torch.utils.data.Dataset` database instance that implements the ``__len__`` and ``__getitem__`` functions.
This class is responsible for loading the image and parsing the x and y values in the image file name.
Since we implemented the ``torch.utils.data.Dataset`` class, we can use all of the torch data utilities, and we hardcoded some conversions (such as color jitter) in the dataset.
We set the random horizontal flip to optional (if you want to follow an asymmetrical path, such as a road), it doesn't matter if Jetbot follows a certain convention, you can enable flips to augment the dataset.

```python
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], [0.229, 0.224, 0.225])

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

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

Yahboom

Split the data set into a training set and a test set that will be used to verify the accuracy of the model we are training:

Split the data set into a training sets and a test sets

Yahboom

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

```python
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])
```

We use the ``DataLoader`` class to load data in bulk, shuffle data, and allow multiple child processes to be used.
In this example, we use a data batch size of 64. The batch size will be based on the memory available to the GPU, which can affect the accuracy of the model.

Run the following cell code to create the training set data loader and test set data loader:

```
# Train sets
train_loader = torch.utils.data.DataLoader(
    train_dataset,
    batch_size=16,
    shuffle=True,
    num_workers=4
)

# Test sets
test_loader = torch.utils.data.DataLoader(
    test_dataset,
    batch_size=16,
    shuffle=True,
    num_workers=4
)
```

Yahboom

The ResNet-18 model we use is based on PyTorch TorchVision. In the process of “migration learning”, also called “transfer learning”, we can reuse a pre-trained model (training millions of images) for one possible A new task with much less data available.
For more information, please visit ResNet-18:
Https://github.com/pytorch/vision/blob/master/torchvision/models/resnet.py

More details about transfer learning (requires science online):
https://www.youtube.com/watch?v=yofjFQddwHE
And transfer it to the GPU via "CUDA":

```
model = models.resnet18(pretrained=True)
```

The ResNet model is fully connected (fc) to the final layer with 512 as `in_features`, we will train the regression, so `out_features` as 1

Finally, we transfer the model to the GPU for execution.

Yahboom

```
model.fc = torch.nn.Linear(512, 2)
device = torch.device('cuda')
model = model.to(device)
```

Then, we can train the regression model we need to use. Here I set the value of NUM_EPOCHS to 50, that is, we trained 50 times. If there is a loss reduction situation, we will save the best model:

```
# NUM_EPOCHS = 70
NUM_EPOCHS = 50
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
```

Yahboom

Once the model is trained, it will generate a ``best_steering_model_xy.pth`` file, which we will use in the autopilot routine for reasoning.

The corresponding complete source code is located:
/home/jetbot/Notebook/17.Autopilot-Basic/train model.ipynb

3. Implementation of motion algorithm

Here we did not use our own PID driver, but a proportional/differential control (PD) controller used to meet our requirements:

```
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_dgai
n_slider.value
angle_last = angle

steering_slider.value = pid + steering_bias_slider.value

#PID+ base speed + Gain
```

```
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)
```

(Base Speed)speed_gain_slider
(P)steering_gain_slider
(D)steering_dgain_slider
(base steering value) steering_bias_slider

We can adjust these four values through the slider to get our Jetbot to drive to the best condition.

4. Autopilot on the track with trained neural network model

First, we need to load the Resnet-18 neural network that has been used many times in this previous course:

```
import torchvision
import torch

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

Yahboom

Then, import the package we need to use and create the relevant instance. To facilitate debugging, I added the line that used the handle to move the Jetbot.

! Note: Turn on the automatic driving and run the corresponding code to close the process of the handle control. The control effect on Jetbot is contradictory, so they are can't run at the same time!

```
from servoserial import ServoSerial
import threading
# 杀掉线程
import inspect
import ctypes
import ipywidgets.widgets as widgets
from IPython.display import display
import time

controller = widgets.Controller(index=0)
display(controller)
```

Yahboom

Next, load the trained model ``best_steering_model_xy.pth`` and transfer it to the GPU for calculation:

```
model.load_state_dict(torch.load('best_steering_model_xy.pth'))
```

Because the model weights are on the CPU memory, then the following code is still passed to the GPU device as always.

```
device = torch.device('cuda')
model = model.to(device)
model = model.eval().half()
```

After the above code is executed, we have loaded the model, but there is a small problem.

In order for the format of our training model to exactly match the format of the camera, we need to do some preprocessing.

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

```
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, ...]
```

Yahboom

Then, the camera's screen is displayed in real time, and the angle of the Jetbot is adjusted to the angle of the automatic driving.

```
from IPython.display import display
import ipywidgets
import traitlets
from jetbot import Camera, bgr8_to_jpeg
from servoserial import ServoSerial

camera = Camera()
servo_device = ServoSerial()

image_widget = ipywidgets.Image()

traitlets.dlink((camera, 'value'), (image_widget, 'value'), transform=bgr8_to_jpeg)

display(image_widget)

def camservoInitFunction():
    global leftrightpulse, updownpulse
    leftrightpulse = 2048
    updownpulse = 2048
    servo_device.Servo_serial_control(1, 2048)
    time.sleep(0.1)
    servo_device.Servo_serial_control(2, 1300)
```

Yahboom

Create an instance robot that controls the Jetbot motion.

```
from jetbot import Robot

robot = Robot()
```

Now, we will define the slider to control the Jetbot.
(Tips: We have configured initial values for the sliders. These initial values apply to our official Yahboom map, but if you are training on your own different road maps, these values may not apply to your dataset, so Please increase or decrease the slider according to your settings and environment)

1) **Speed control (speed_gain_slider):** To start Jetbot, add ``speed_gain_slider``
2) **Steering gain control (steering_gain_sloder):** If you see that Jetbot is

spinning, you need to reduce ``steering_gain_slider`` until it becomes smooth.
3)Steering bias control (steering_bias_slider): If you see Jetbot leaning towards the far right or extreme left of the track, you should control this slider until Jetbot starts tracking the line or track at the center.

(Note: When you slide the related slider mentioned above, you should not move the slider value very quickly to get a smooth Jetbot road following behavior. You should adjust the motion parameter by moving the slider value gently.)

```
speed_gain_slider = ipywidgets.FloatSlider(min=0.0, max=1.0, step=0.01, value=0.75,description='speed gain')
steering_gain_slider = ipywidgets.FloatSlider(min=0.0, max=1.0, step=0.01, value=0.33, description='steering gain
steering_dgain_slider = ipywidgets.FloatSlider(min=0.0, max=0.5, step=0.001, value=0.12, description='steering kd
steering_bias_slider = ipywidgets.FloatSlider(min=-0.3, max=0.3, step=0.01, value=0, description='steering bias')

display(speed_gain_slider, steering_gain_slider, steering_dgain_slider, steering_bias_slider)
```

The x and y sliders will display the predicted x, y values. The steering slider will display our estimated steering value. This value is not the actual angle of the target, but an almost proportional value.
When the actual angle is ``0``, this is 0, which will increase/decrease as the actual angle increases/decreases:

```
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)
```

Next, we'll create a function that will be called when the camera's value changes. This function will perform the following steps:
1) Preprocess camera image
2) Perform a neural network
3) Calculate the approximate steering value
4) Control the motor using proportional/differential control (PD)

```
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_dgain_slider.value
    angle_last = angle

    steering_slider.value = pid + steering_bias_slider.value

    #PID+基础速度+增益
    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})
```

Yahboom

We have created a neural network execution function, but now we need to attach it to the camera for processing.

(Tips: This code will make the robot move!! Please place the Jetbot robot on the map you have trained before. If the data you collected and the model are well trained, you will see Jetbot running smoothly on the road.)

```
camservoInitFunction()
camera.observe(execute, names='value')
```

Yahboom

If your Jetbot functions properly, it will generate new commands for each new camera frame. Now you can place the Jetbot on a track that has collected data and see if it can track the track. If you want to stop this behavior, you can unload the binding of this callback function by executing the code in the following cell:

```
camera.unobserve(execute, names='value')
time.sleep(0.1)
robot.stop()
```

Yahboom

You can run the following code to open the thread with the handle to remotely control Jetbot.

! Note: When using the handle, please run the code in the above cell to stop the automatic driving function of Jetbot.

```
def jetbot_motion():
    count1 = count2 = count3 = count4 =  count5 = 0
    while 1:
        #小车左右DC motor
        if controller.axes[1].value <= 0:
            robot.set_motors(-controller.axes[1].value + controller.axes[0].value, -controller.axes[1].value - co
            time.sleep(0.01)
        else:
            robot.set_motors(-controller.axes[1].value - controller.axes[0].value, -controller.axes[1].value + co
            time.sleep(0.01)

# thread1 = threading.Thread(target=jetbot_motion)
# thread1.setDaemon(False)
# thread1.start()

def _async_raise(tid, exctype):
    """raises the exception, performs cleanup if needed"""
    tid = ctypes.c_long(tid)
    if not inspect.isclass(exctype):
        exctype = type(exctype)
    res = ctypes.pythonapi.PyThreadState_SetAsyncExc(tid, ctypes.py_object(exctype))
    if res == 0:
        raise ValueError("invalid thread id")
    elif res != 1:
        # """if it returns a number greater than one, you're in trouble,
        # and you should call it again with exc=NULL to revert the effect"""
        ctypes.pythonapi.PyThreadState_SetAsyncExc(tid, None)

def stop_thread(thread):
    _async_raise(thread.ident, SystemExit)
```

```
thread1 = threading.Thread(target=jetbot_motion)
thread1.setDaemon(False)
thread1.start()
```

```
stop_thread(thread1)
```

Yahboom

The corresponding complete source code is located:
/home/jetbot/Notebook/17.Autopilot-Basic/Autopilot-Basic.ipynb

5. Auto-driving pedestrians (multi-object optional) detect parking

Based on the Autopilot-Basic Edition, we tried to port the object detection function in the object following example:

Load the object detection model and add related algorithm methods. The code is as shown below:

```
from jetbot import ObjectDetector
global object_model
object_model = ObjectDetector('ssd_mobilenet_v2_coco.engine')

def detection_center(detection):
    """Calculate the center x, y coordinates of the object"""
    bbox = detection['bbox']
    center_x = (bbox[0] + bbox[2]) / 2.0 - 0.5
    center_y = (bbox[1] + bbox[3]) / 2.0 - 0.5
    return (center_x, center_y)

def norm(vec):
    """Calculate the length of a two-dimensional vector"""
    return np.sqrt(vec[0]**2 + vec[1]**2)

def closest_detection(detections):
    """Find the detection closest to the center of the image"""
    #先清除缓存
    closest_detection = None
    for det in detections:
        center = detection_center(det)
        if closest_detection is None:
            closest_detection = det
        elif norm(detection_center(det)) < norm(detection_center(closest_detection)):
            closest_detection = det
    return closest_detection
```

Yahboom

This code is a bit different from the real-time display camera code, because the data is processed to return the image to the display component display, so the traitles dLink camera value and display component values are not used in the code that displays the image.

```
from IPython.display import display
# import ipywidgets
import ipywidgets.widgets as widgets
import traitlets
from jetbot import Camera, bgr8_to_jpeg
from servoserial import ServoSerial

# camera = Camera()
camera = Camera.instance(width=300, height=300)
servo_device = ServoSerial()

# image_widget = ipywidgets.Image()
# traitlets.dlink((camera, 'value'), (image_widget, 'value'), transform=bgr8_to_jpeg)
# display(image_widget)

image_widget = widgets.Image(format='jpeg', width=300, height=300)
display(image_widget)


def camservoInitFunction():
    global leftrightpulse, updownpulse
    leftrightpulse = 2048
    updownpulse = 2048
    servo_device.Servo_serial_control(1, 2048)
    time.sleep(0.1)
    servo_device.Servo_serial_control(2, 1300)
```

Yahboom

Use the following code to detect, object detection, detected objects:

```
detections = object_model(camera.value)
print(detections)
```

Yahboom

Then, we implement the detection of the object by adding the following code to the autopilot-based version of the motion control code.

If we detect the object corresponding to the label we set, we will let Jetbot stop and restart the automatic driving when the object no longer appears in the field of view:

```
# Calculate all detected objects
detections = object_model(image)

# Draw all detected objects
  for det in detections[0]:
      bbox = det['bbox']
      cv2.rectangle(image, (int(300 * bbox[0]), int(300 * bbox[1])), (int(300 * bbox[2]), int(300 * bbox[3])), (255, 0, 0), 2)

#Select the object you want to track, select the value of label_widget, 1 is person
# select detections that match selected class label
matching_detections = [d for d in detections[0] if d['label'] == 1]

#Mark the object to be tracked with green lines.
# get detection closest to center of field of view and draw it
det = closest_detection(matching_detections)
if det is not None:
      bbox = det['bbox']
      cv2.rectangle(image, (int(300 * bbox[0]), int(300 * bbox[1])), (int(300 * bbox[2]), int(300 * bbox[3])), (0, 255, 0), 4)
    ''' Stop the current Jetbot movement if it detects an object on the road that needs to be avoided '''
    robot.stop()
```

The corresponding complete source code is located:
/home/jetbot/Notebook/18.Autopilot Pedestrian detects parking/Autopilot Pedestrian detects parking.ipynb