5.4 Automatic avoid

1. Collect data through Jetbot robot car

First, initialize the camera and display in real time. The image data we collected and its properties are consistent with the currently set camera display image.

Our neural network adopt 224 x 224 pixel images as input. So we set the camera to this size to minimize the file size and minimize the data set. (We have tested this pixel for this task.) In some cases, it is best to use a larger image size when collecting data, and then reduce it to the required size when processing.

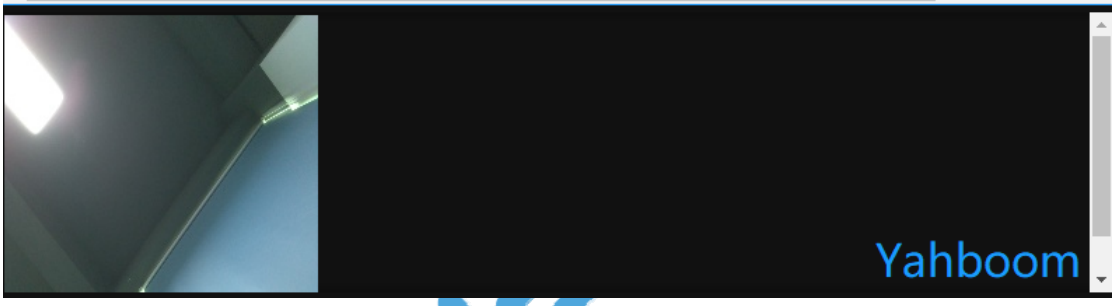
```
import traitlets
import ipywidgets.widgets as widgets
from IPython.display import display
from jetbot import Camera, bgr8_to_jpeg

camera = Camera.instance(width=224, height=224)

image = widgets.Image(format='jpeg', width=224, height=224)  # this width and height doesn't necessarily have

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

display(image)
```



Then, we need to create some directories to store the data. We will create a folder called dataset with two sub-folders, which is free and blocked, for sorting the images of each scene.

```
import os

blocked_dir = 'dataset/blocked'
free_dir = 'dataset/free'

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


If you are running the cell code at second time, the previously created data store folder already exists and you will be prompted with 'Directories not created becasue they already exist'

If you refresh the Jupyter file browser on the left, you can see that these directories appear.

Next, we need to create and display some buttons that will be used to save a snapshot for each class tag. We'll also add some text boxes that will display

the number of images for each category we've collected so far.

This is useful because we want to make sure that we collect as many "free" images as "block" images. It also helps to understand how many images we have collected in total.

```
button_layout = widgets.Layout(width='128px', height='64px')
free_button = widgets.Button(description='add free', button_style='success', layout=button_layout)
blocked_button = widgets.Button(description='add blocked', button_style='danger', layout=button_layout)
free_count = widgets.IntText(layout=button_layout, value=len(os.listdir(free_dir)))
blocked_count = widgets.IntText(layout=button_layout, value=len(os.listdir(blocked_dir)))

display(widgets.HBox([free_count, free_button]))
display(widgets.HBox([blocked_count, blocked_button]))
```

After running the above cell code, the following two buttons and counters are displayed:

But now these two buttons are still only displayed, no events have been bound, we have to attach a function to save the image for each category of button ' ' on_click ' ' event. We will save the value of ' ' Image ' ' (not the camera) because it is already in compressed JPEG format!

To make sure that no filenames are repeated (even on different machines!), we'll use the ' 'uiid' package in python, which defines the ' 'uiid1" method to generate a unique identifier. This unique identifier is generated from information such as current time and machine address.

```
from uuid import uuid1

def save_snapshot(directory):
    image_path = os.path.join(directory, str(uuid1()) + '.jpg')
    with open(image_path, 'wb') as f:
        f.write(image.value)

def save_free():
    global free_dir, free_count
    save_snapshot(free_dir)
    free_count.value = len(os.listdir(free_dir))

def save_blocked():
    global blocked_dir, blocked_count
    save_snapshot(blocked_dir)
    blocked_count.value = len(os.listdir(blocked_dir))

# attach the callbacks, we use a 'lambda' function to ignore the
# parameter that the on_click event would provide to our function
# because we don't need it.
free_button.on_click(lambda x: save_free())
blocked_button.on_click(lambda x: save_blocked())
```

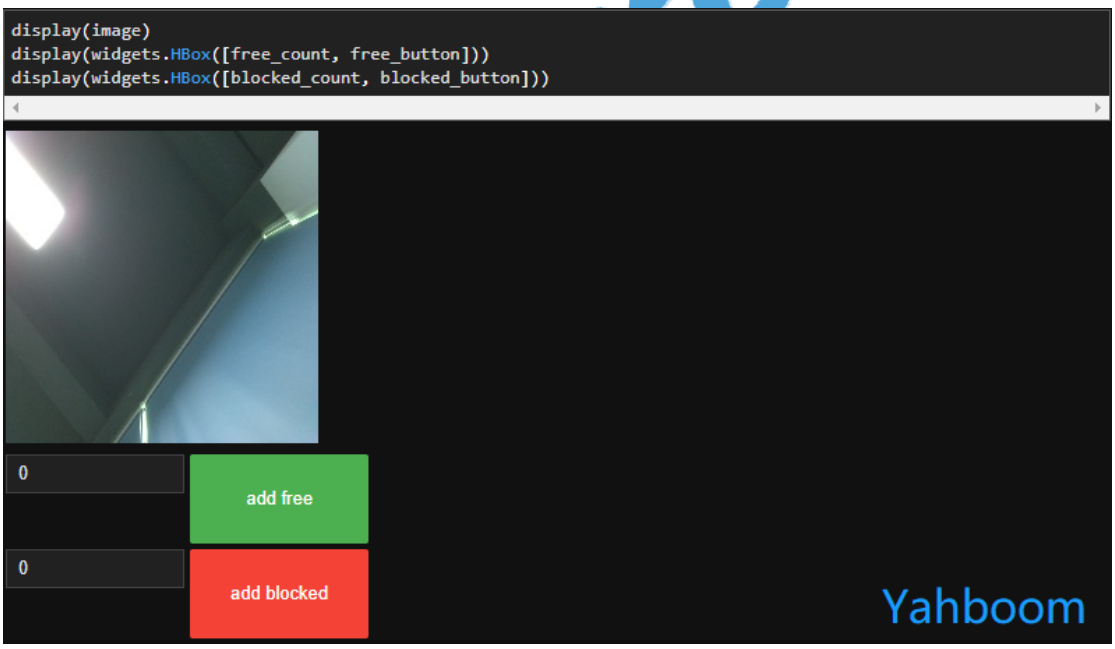
Now, the button above can already save the image to the "free" and "blocked" directories. You can view these files using the Jupyter left directory file browser!

Now continue to collect some data:
1. Place the robot in a scene where it is blocked and press "add blocked".
2. Place the robot in a free scene and press "add free".
3. Repeat 1, 2
(Reminder: You can move the widget to a new window by right-clicking the cell and clicking Create new View for Output.)

Here are some tips for tagging data:
Try different directions
2. Try different lighting
3. Try different object/conflict types; walls, rock shelves, objects
4. Try floors/objects of different textures; patterned, smooth, glass, etc.

It's important to get a variety of data (as described in the tips above), not just a lot of data, and you might need at least 100 images per class (this is not science, just a useful trick).

After running the cell code below, the image and button will be displayed and you can start collecting data:



When collect enough data, we need to copy this data to our GPU platform for training, but our Jetbot is powerful enough that we don't have to wait too long for Jetbot training.
The corresponding complete source code is located:
/home/jetbot/Notebook/13.Automatic avoid/Data collection.ipynb

2. Train the neural network model

Here, we will train our image classifier to detect two classes 'free' and 'blocked', which we will use to avoid collisions.
First, we need to import the relevant libraries used by torch and torchvision:
Code as shown below:

```
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
```


We create an instance of the dataset using the ImageFolder dataset class in the torchvision.datasets library.
There is an additional torchvision.transforms library for converting data to prepare for the training model.
Code as shown below:

```
dataset = datasets.ImageFolder(
    'dataset',
    transforms.Compose([
        transforms.ColorJitter(0.1, 0.1, 0.1, 0.1),
        transforms.Resize((224, 224)),
        transforms.ToTensor(),
        transforms.Normalize([0.485, 0.456, 0.406], [0.229, 0.224, 0.225])
    ])
)
```


We divide the data set into a training set and a test set. The test set is used to help us verify the accuracy of the trained model:
Code as shown below:

Split the data sets into training sets and test sets

Next, we split the data set into a training set and a test set. The test set will be used to verify the accuracy of our trained models.

```
train_dataset, test_dataset = torch.utils.data.random_split(dataset, [len(dataset) - 10, 10])
```

Next, we need to create a data loader to load the data in bulk. The data loader is divided into two: one is the training data loader and the other is the test data loader:
Code as shown below:

Create a data loader to load data in bulk

We will create two DataLoader instances that provide utilities for shuffling data, generate batch images, and load samples in parallel with multiple tasks.

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

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

Yahboom

Now, we define the neural network that will be trained.
The torchvision library provides a set of pre-trained models that we can use. In a process called migration learning, also called learning, we can reuse pre-trained models (training on millions of images) to get as little data as possible and to do as many tasks as possible. Important features learned in the original training of the pre-training model can be reused for new tasks.
Here we will use another model called the alexnet model. The alexnet model was originally trained for datasets with 1000 class labels, but our dataset has only two class labels!
We will replace the best layer with the latest, untrained layer with only two outputs.
Code as shown below:

The alexnet model was originally trained for datasets with 1000 class labels, but our dataset has only two class labels!

We will replace the best layer with the latest, untrained layer with only two outputs.

Yahboom

```
model.classifier[6] = torch.nn.Linear(model.classifier[6].in_features, 2)
```

Then, transfer the model from the CPU to the GPU via 'CUDA':
Code as shown below:

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

Yahboom

After running the above code, we can start the model training directly with the cell code below.
Code as shown below:

Training neural network

Using the code below, we will begin to train our neural network and save the best performing model after each generation.

> One generation is to run all the data again

```
NUM_EPOCHS = 30
BEST_MODEL_PATH = 'best_model.pth'
best_accuracy = 0.0

optimizer = optim.SGD(model.parameters(), lr=0.001, momentum=0.9)

for epoch in range(NUM_EPOCHS):

    for images, labels in iter(train_loader):
        images = images.to(device)
        labels = labels.to(device)
        optimizer.zero_grad()
        outputs = model(images)
        loss = F.cross_entropy(outputs, labels)
        loss.backward()
        optimizer.step()

    test_error_count = 0.0
    for images, labels in iter(test_loader):
        images = images.to(device)
        labels = labels.to(device)
        outputs = model(images)
        test_error_count += float(torch.sum(torch.abs(labels - outputs.argmax(1))))

    test_accuracy = 1.0 - float(test_error_count) / float(len(test_dataset))
    print('%d: %f' % (epoch, test_accuracy))
    if test_accuracy > best_accuracy:
        torch.save(model.state_dict(), BEST_MODEL_PATH)
        best_accuracy = test_accuracy
```

When the model is trained, we can see the model named best_model.th generated in the catalog. We will use this model for active obstacle avoidance in the next routine.

The corresponding complete source code is located at:

/home/jetbot/Notebook/13.Automatic avoid/train model.ipynb

3. Using a trained neural network model to avoid obstacles

In the above two sections, we trained the data collected by the 'acquisition data' code by running the 'training model' to become the obstacle avoidance model we need.

Now, we will use this model step by step to realize the obstacle avoidance function.

Execute the following code to initialize the PyTorch model:

Code as shown below:

```
import torch
import torchvision

model = torchvision.models.alexnet(pretrained=False)
model.classifier[6] = torch.nn.Linear(model.classifier[6].in_features, 2)
```


Then, we load the model we trained last time. If the model that we train is not ideal, it may be related to the number of data acquisition and the height and angle of the camera. Users can test directly using the trained model in the

image provided by Yahboom.
After downloading, we also need to transfer the model weights from the CPU to the GPU operation via 'CUDA'.
Code as shown below:

Load your own uploaded, `best_model.pth` model that has been trained

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

At present, the model weight calculation is located in the CPU memory. We transfer the model to the GPU through the introduction of our previous tutorial or 'CUDA', and execute the following code to use the GPU.

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


After the loading model is complete.
We need to do some pre-processing to ensure that the image format of our camera is exactly the same as the image format when training the model.
Need to perform the following steps:
1. Convert from BGR to RGB mode
2. Convert from HWC layout to CHW layout
3. Normalize using the same parameters as during training (our camera provides values in the range [0, 255] and trains the loaded image in the range [0, 1], so we need to scale 255.0)
4. Transfer data from CPU memory to GPU memory
5. Add dimensions in bulk
Run the following code to define the pre-processing features:

```
import cv2
import numpy as np

mean = 255.0 * np.array([0.485, 0.456, 0.406])
stdev = 255.0 * np.array([0.229, 0.224, 0.225])

normalize = torchvision.transforms.Normalize(mean, stdev)

def preprocess(camera_value):
    global device, normalize
    x = camera_value
    x = cv2.cvtColor(x, cv2.COLOR_BGR2RGB)
    x = x.transpose((2, 0, 1))
    x = torch.from_numpy(x).float()
    x = normalize(x)
    x = x.to(device)
    x = x[None, ...]
    return x
```


Once the preprocessing function is defined, the image can be converted from a camera format to a neural network input format.

We also need to create a slider that shows the probability that the robot is blocked:
Code as shown below:

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

camera = Camera.instance(width=224, height=224)
image = widgets.Image(format='jpeg', width=224, height=224)
blocked_slider = widgets.FloatSlider(description='blocked', min=0.0, max=1.0, orientation='vertical')

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

display(widgets.HBox([image, blocked_slider]))
```


Create a robot instance responsible for Jetbot motion control:
Code as shown below:

```
from jetbot import Robot

robot = Robot()
```


Then,we need to create a function that calls the function whenever the value of the camera changes. This feature will perform the following steps:

1. Pre-process camera image
2. Perform a neural network
3. When the neural network output indicates that we are blocked, we will turn left, otherwise we will move on.

Code as shown below:

```
import torch.nn.functional as F
import time

def update(change):
    global blocked_slider, robot
    x = change['new']
    x = preprocess(x)
    y = model(x)

    # we apply the `softmax` function to normalize the output vector so it sums to 1 (which makes it a probabi
    y = F.softmax(y, dim=1)

    prob_blocked = float(y.flatten()[0])

    blocked_slider.value = prob_blocked

    #if prob_blocked < 0.5:
        #robot.forward(0.4)
    #else:
        #robot.left(0.4)
    if prob_blocked < 0.78:
        robot.stop()
#         robot.forward(0.7)
    else:
        robot.stop()
#         robot.left(0.7)

    time.sleep(0.001)

update({'new': camera.value})  # we call the function once to intialize
```


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

Tips: The robot will start moving at this time! Make sure your Jetbot robot is in

a movable area to avoid falling damage!!!

```
camera.observe(update, names='value')  # this attaches the 'update' function to the 'value' traitlet of our ca
```

After running the above code block, Jetbot starts generating new commands for each detected picture.

First, we need to put the robot on the ground and observe its reaction when it encounters an obstacle.
If you want to stop performing autonomous obstacle avoidance, you can cancel by executing the following code.

```
camera.unobserve(update, names='value')
time.sleep(1)
robot.stop()
```

Execute the code shown below, you can cancel the camera connection, just do not push the video data stream to the browser, but the Jetbot camera is still working.

```
camera_link.unlink()  # don't stream to browser (will still run camera)
```

If you need to re-display refresh, run the following code:

```
camera_link.link()  # stream to browser (wont run camera)
```

The corresponding complete source code is located at:
/home/jetbot/Notebook/13.Automatic avoid/Automatic avoid with trained model.ipynb