

Collision Avoidance

In this example, we'll collect an image classification dataset that will be used to help keep JetBot safe! We'll teach JetBot to detect two scenarios free and blocked. We'll use this AI classifier to prevent JetBot from entering dangerous territory.

Step 1: Collect data on JetBot

- Access JetBot by going to https://<jetbot_ip_address>:8888, navigate to ~/Notebooks/collision_avoidance/.
- Open data_collection.ipynb file and following notebook.
- After running the program, the interface as shown in the figure appears, put the car in a different position, and click "add free" if there is no obstacle in front of the car. If there is an obstacle in front of the car, please click "add blocked".
- The captured pictures will be saved in the dataset folder, and as many pictures of various situations as possible will be taken. You can try different orientations, brightness, object or collision types (walls, ledges, etc.), and can try untextured floors/objects (patterned, smooth, glass, etc.).

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



- The more scene data the car collects, the better the obstacle avoidance effect will be. Therefore, it is very important to obtain as much different

data as possible for the obstacle avoidance effect. Generally, at least 100 pictures are required for each situation.

- Finally, run the program to package the pictures. After packaging, a dataset.zip compressed file will appear in the current directory.

```
[7]: !zip -r -q dataset.zip dataset
```

Step 2. Train neural network

- Access JetBot by going to https://<jetbot_ip_address>:8888, navigate to ~/Notebooks/collision_avoidance/
- Open and follow the train_model.ipynb notebook.
- If you already have the dataset.zip file you just compressed, you do not need to run this statement to decompress it, otherwise you will be prompted to overwrite the existing file.



The screenshot shows the JetBot JupyterLab interface. On the left, the file browser displays a list of files: 'dataset' (an hour ago), 'data_collection.ipynb' (an hour ago), 'live_demo.ipynb' (6 hours ago), 'train_model.ipynb' (6 minutes ago), 'test_model.pth' (11 minutes ago), and 'dataset.zip' (an hour ago). The 'train_model.ipynb' file is selected. The main area shows the code cells of the notebook. The first cell contains import statements for torch, torch.optim, torch.nn.functional, torchvision, torchvision.datasets, torchvision.models, and torchvision.transforms. The second cell is titled 'Upload and extract dataset' and contains instructions to upload the 'dataset.zip' file and extract it using the command `!unzip -q dataset.zip`. The command is highlighted with a red box. Below the command, there is a placeholder for a file path: `replace dataset/blocked/93d81b3c-8e75-11e9-9231-72b5f773b75d.jpg [y]es, [n]o, [A]ll, [U]nion, [r]ename`. The third cell contains the instruction: 'You should see a folder named dataset appear in the file browser.'

- Finally, run the program to train the neural network, and the running time is relatively long. After the training is completed, a best_mode.pth file will appear in the current directory.

The screenshot shows a Jupyter Notebook with a file named `train_model.ipynb`. The code in the notebook defines a training loop for 30 epochs. It uses an SGD optimizer with a learning rate of 0.001 and momentum of 0.9. The script tracks the test accuracy and saves the best model. The output view on the right shows the test accuracy for each epoch, ranging from 0.800000 to 0.920000.

```
[9]: NBP_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(NBP_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
    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
```

Output View:

```
0: 0.900000
1: 0.820000
2: 0.860000
3: 0.880000
4: 0.860000
5: 0.900000
6: 0.880000
7: 0.860000
8: 0.940000
9: 0.900000
10: 0.840000
11: 0.860000
12: 0.900000
13: 0.840000
14: 0.840000
15: 0.940000
16: 0.900000
17: 0.880000
18: 0.920000
19: 0.920000
20: 0.920000
21: 0.920000
22: 0.880000
23: 0.860000
24: 0.920000
25: 0.920000
26: 0.920000
27: 0.880000
28: 0.920000
29: 0.860000
```

- When the training of the model is going on then the value of the accuracy means the value 1.000000 means it is the best accuracy also 0.9 is also considered as the best accuracy. From 0.5 to 0.0 considered as worst accuracy

Step 3. Automatic Obstacle Avoiding

- Access JetBot by going to https://<jetbot_ip_address>:8888, navigate to `~/Notebooks/collision_avoidance/`
- Open and follow the `live_demo.ipynb` notebook.
- After running the program, the camera live image and a slider are displayed. Intermodulation represents the probability of encountering an obstacle, 0.00 means that there is no obstacle ahead, and 1.00 means that the obstacle ahead needs to be turned to avoid.

The screenshot shows a Jupyter Notebook with a file named `live_demo.ipynb`. The code in the notebook shows how to interact with the robot's camera and the obstacle avoidance slider. The output view on the right shows a live camera feed of a robot on a green surface and a slider labeled 'blocked' with a value of 0.00. A red arrow points to the slider.

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

Perhaps you want the robot to run without streaming video to the browser. You can
unlink the camera as below.

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

To continue streaming call the following.

[ ]: camera_link.link() # stream to browser (won't run camera)
```

Output View:

blocked

0.00

- Here, adjust the speed a little to avoid hitting the obstacles too fast. If obstacle avoidance cannot be achieved in some places, it is recommended to collect more data.

```
[ ]: 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 probability)
    y = F.softmax(y, dim=1)

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

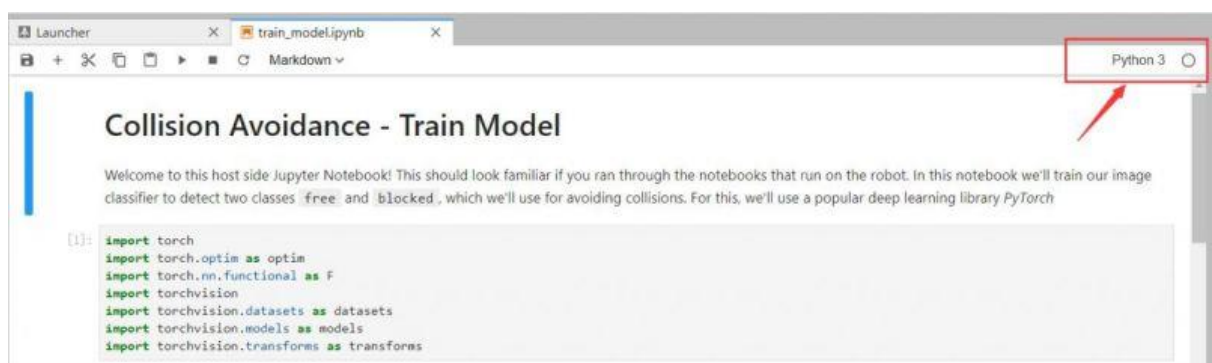
    blocked_slider.value = prob_blocked

    if prob_blocked < 0.5:
        robot.forward(0.2)
    else:
        robot.left(0.1)

    time.sleep(0.001)

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

- [Important Note] When the Jetbot gets start then execute the next cell as it connects the camera so it will avoid the obstacles otherwise it will not stop as the camera instance needs to connect then you can see the 1.00 to the blocked image and 0.00 to free image
- [Note] Some statements may take a long time to run. There is a program running promptly in the upper right corner of JupyterLab. When the small dot is black, it means the program is running, and white means it is idle.



The model is trained by using red tape so that it will recognize the obstacle as a red tape

If the model training is more then you will get more accuracy.