ECE140A Lab 2 - Part 1: Collision Avoidance

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

Collision Avoidance

In lab 1, you assembled the JetBot and moved it around in a square. Hopefully, now you understand how make the robot move around by interacting with the `jetbot` library.

What would be more interesting, however, is to move the JetBot around autonomously! While complete autonomy is a very difficult task, we can break it down into easier subproblems. Arguably, one of the most important subproblems to solve is to prevent the robot from entering dangerous situations: bumping into things or falling off cliffs! Let's call this *collision avoidance*.

We will use deep learning with camera input to solve collision avoidance. We will do this in a simple way:

1. Data collection: We'll manually place the robot in situations where its "safety bubble" is violated, and label the corresponding camera images as `blocked` . And we'll also place the robot in scenarios where it's safe to move forward a bit, and label the corresponding images as `free` .
2. Training/Fine-tuning: Using the images collected in the previous step, we'll train a deep neural network to predict if it is `blocked` or `free` from the current image in the camera feed. Instead of training a network from scratch, we'll "fine-tune" or improve the model already trained by the folks at NVIDIA.
3. Testing and improving failure cases: We'll run our JetBot with collision avoidance to see how it performs. Taking note of scenarios where the robot doesn't avoid collision as expected, we'll collect some more data from those scenarios.

Repeat steps 2 and 3 as many times (not zero!) as you like.

> There's a lot of "advanced" code in this notebook. You don't need to understand it all. We'll just say, in words, what the code does, and you can assume that it works as we say.

Data Collection

Data collection sounds like the most tedious thing ever, and it can be. However, because we're using a pretrained neural network (more on that later), we don't need to collect a lot of data. We will just collect 50 images or more per class (100 or more images in total). (It will just take 10 minutes)

Later, the TAs will combine the data collected by all the students to make a super collision avoider!

Display live camera feed

Let's initialize and display our 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.

> It's not important to understand how to code below works.

In [1]:
```
import traitlets
import ipywidgets.widgets as widgets
from IPython.display import display
```

In [19]:
```
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 heig
ht doesn't necessarily have to match the camera

camera_link = traitlets.dlink((camera, 'value'), (image, 'value'), transform=bgr8_
to_jpeg)

display(image)
```

We'll now create a few directories where we'll store our data: a folder `dataset` that will contain two subfolders `free` and `blocked`, where we'll place images for each scenario.

In [20]:
```
import os

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

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

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

Refresh the Jupyter file browser on the left to see those directories appear. Next, let's create some buttons that we'll use to save snapshots. Click `free` if the current image corresponds to the robot having some space to move forward. Click `blocked` if there is an obstacle in front of the robot such that it will not be able to move any further or rotate in place. It's better to be conservative here, that is, if you're not sure if a situation is `free` or `blocked` it's better to label it as `blocked`.

> Again, you don't have to understand the following code.
>
> The counts displayed in the widgets can be off by 1 (or sometimes even more) because they count all the files in the blocked and free folders (not just images). Continue to the next part only if **our checker** says you're good!

In [21]:
```
button_layout = widgets.Layout(width='128px', height='64px')
free_button = widgets.Button(description='add free', button_style='success', layou
t=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_di
r)))
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]))

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())
```

In [22]:
```
from checker import check_collected_images
check_collected_images('dataset')
```

```
You are good to go!
```

Developing an understanding of the Neural Network

Below are some resources which introduce the concept of neural networks :

> https://towardsdatascience.com/a-gentle-introduction-to-neural-networks-series-part-1-2b90b87795bc (https://towardsdatascience.com/a-gentle-introduction-to-neural-networks-series-part-1-2b90b87795bc)
>
> https://victorzhou.com/blog/intro-to-neural-networks/ (https://victorzhou.com/blog/intro-to-neural-networks/)

What is PyTorch?

- An open source machine learning framework that accelerates the path from research prototyping to production deployment.
- For more reading please visit

> http://pytorch.org/tutorials/beginner/deep_learning_60min_blitz.html (http://pytorch.org/tutorials/beginner/deep_learning_60min_blitz.html)

Getting Started

- NOTE: Make sure that you have numpy and pytorch installed before importing the python frameworks (they are on the JetBot).

In [2]:
```
import numpy as np
import torch
```

Tensors

- Torch tensors are similar to NumPy ndarrays, with the addition being that they can also be used on a GPU to accelerate computing.
- The documentation for tensors is available here

> http://pytorch.org/docs/stable/tensors.html (http://pytorch.org/docs/stable/tensors.html)

1. Construct a 5 × 3 tensor and print it with the following commands

In [24]:
```
x = torch.Tensor(5, 3)
print(x)
```

```
tensor([[15.7713,  0.0000,  0.0000],
        [ 0.0000,  0.0000,  0.0000],
        [ 0.0000,  0.0000, 15.5088],
        [ 0.0000,  0.0000,  0.0000],
        [15.4296,  0.0000, 15.5117]])
```

1. Construct a randomly initialized tensor

In [25]:
```
y = torch.rand(5, 3)
print(y)
```

```
tensor([[0.4801, 0.5936, 0.5651],
        [0.8416, 0.8905, 0.8101],
        [0.6504, 0.8356, 0.4934],
        [0.6305, 0.6600, 0.3203],
        [0.1018, 0.0820, 0.7809]])
```

1. We can also directly initialize tensors with prescribed values. Create the following two tensors. What are their shapes ?

In [26]:
```
x = torch.Tensor([[-0.1859, 1.3970, 0.5236],
[ 2.3854, 0.0707, 2.1970],
[-0.3587, 1.2359, 1.8951],
[-0.1189, -0.1376, 0.4647],
[-1.8968, 2.0164, 0.1092]])
y = torch.Tensor([[ 0.4838, 0.5822, 0.2755],
[ 1.0982, 0.4932, -0.6680],
[ 0.7915, 0.6580, -0.5819],
[ 0.3825, -1.1822, 1.5217],
[ 0.6042, -0.2280, 1.3210]])
x
```

Out[26]:
```
tensor([[-0.1859,  1.3970,  0.5236],
        [ 2.3854,  0.0707,  2.1970],
        [-0.3587,  1.2359,  1.8951],
        [-0.1189, -0.1376,  0.4647],
        [-1.8968,  2.0164,  0.1092]])
```

1. Adding two tensors is fairly straightforward. There are multiple syntaxes for this operation. Sum x and y (from question 4) as follows:

In [27]:
```
print(x + y)
print(torch.add(x, y))
print(x.add(y))
torch.add(x, y, out=x)
print(x)
```

```
tensor([[ 0.2979,  1.9792,  0.7991],
        [ 3.4836,  0.5639,  1.5290],
        [ 0.4328,  1.8939,  1.3132],
        [ 0.2636, -1.3198,  1.9864],
        [-1.2926,  1.7884,  1.4302]])
tensor([[ 0.2979,  1.9792,  0.7991],
        [ 3.4836,  0.5639,  1.5290],
        [ 0.4328,  1.8939,  1.3132],
        [ 0.2636, -1.3198,  1.9864],
        [-1.2926,  1.7884,  1.4302]])
tensor([[ 0.2979,  1.9792,  0.7991],
        [ 3.4836,  0.5639,  1.5290],
        [ 0.4328,  1.8939,  1.3132],
        [ 0.2636, -1.3198,  1.9864],
        [-1.2926,  1.7884,  1.4302]])
tensor([[ 0.2979,  1.9792,  0.7991],
        [ 3.4836,  0.5639,  1.5290],
        [ 0.4328,  1.8939,  1.3132],
        [ 0.2636, -1.3198,  1.9864],
        [-1.2926,  1.7884,  1.4302]])
```

1. To reshape a tensor, you can use torch.view. Interpret the effect of each of these instructions. What does the -1 mean?

In [28]:
```
x = torch.randn(4, 4)
y = x.view(16)
z = x.view(-1, 8)
print(x.size(), y.size(), z.size())
```

```
torch.Size([4, 4]) torch.Size([16]) torch.Size([2, 8])
```

> Go through the documentation and get familiar with other operations as well

Autograd: automatic differentiation

- A central feature in PyTorch is the autograd package. It provides tools performing automatic differentiation for all operations based on Torch tensors. This means that the gradients of such operations will not require to be explicitly programmed.
- It uses a define-by-run framework that computes these gradients dynamically during runtime.

Once you have developed a basic understanding of Neural Networks you can look into the following pyTorch tutorials

> https://pytorch.org/tutorials/beginner/blitz/neural_networks_tutorial.html#sphx-glr-beginner-blitz-neural-networks-tutorial-py (https://pytorch.org/tutorials/beginner/blitz/neural_networks_tutorial.html#sphx-glr-beginner-blitz-neural-networks-tutorial-py)
>
> https://pytorch.org/tutorials/beginner/blitz/cifar10_tutorial.html#sphx-glr-beginner-blitz-cifar10-tutorial-py (https://pytorch.org/tutorials/beginner/blitz/cifar10_tutorial.html#sphx-glr-beginner-blitz-cifar10-tutorial-py)

NOTE : The required output from the above exercise is that you are able to understand and run the code of the above two notebooks as it will be used later in the Lab.

Let's Go Deeper: Training a Powerful Deep Neural Network

Now you know a little bit about different building blocks (layers) of neural networks and can construct a network with PyTorch. The network you wrote is a toy model and can work well for relatively simple tasks. However, for more complex tasks like collision avoidance and path following, we'll need a more powerful neural network.

In [3]:
```
import torch
import torch.nn as nn
import torch.nn.functional as F
```

Complete the definition of `DeepDeepNet` using the given image.

> Don't be intimidated by the size of the diagram; each block corresponds to just 1-2 lines of code! Pro-tip: download the image and look at it in another window as you write the code.

DeepDeepNet.png

In [4]:
```
class DeepDeepNet(nn.Module):
    def __init__(self):
        super(DeepDeepNet, self).__init__()
        self.conv1 = nn.Conv2d(3, 64, 11, 4 ,2)#done
        self.relu1 = nn.ReLU()#done
        self.mp1 = nn.MaxPool2d(3,2,0)#self.relu1,(3,2,0)#done
        self.conv2 = nn.Conv2d(64,192,5,1,2)#done
        self.relu2 = nn.ReLU()#done
        self.mp2 = nn.MaxPool2d(3,2,0)#self.relu2,(3,2,0)#done
        self.conv3 = nn.Conv2d(192,384,3,1,1)#done
        self.relu3 = nn.ReLU()#done
        self.conv4 = nn.Conv2d(384,256,3,1,1)#done
        self.relu4 = nn.ReLU()#done
        self.conv5 = nn.Conv2d(256,256,3,1,1)#done
        self.relu5 = nn.ReLU()#done
        self.mp3 = nn.MaxPool2d(3,2,0)#done
        self.avgpool = nn.AdaptiveAvgPool2d((6, 6))#done
        self.dropout1 = nn.Dropout(0.5)#done
        self.linear1 = nn.Linear(9216,4096)#done
        self.relu6 = nn.ReLU()#done
        self.dropout2 = nn.Dropout(0.5)#done
        self.linear2 = nn.Linear(4096,4096)#done
        self.relu7 = nn.ReLU()#done
        self.linear3 = nn.Linear(4096, 2)#done

    def forward(self, x):
        x = self.relu1(self.conv1(x))
        x = self.mp1(x)

        x = self.relu1(self.conv2(x))
        x = self.mp2(x)

        x = self.relu1(self.conv3(x))
        x = self.relu1(self.conv4(x))
        x = self.relu1(self.conv5(x))
        x = self.mp3(x)

        # Complete code here such that x is the output just before the avgpool lay
er
        # after "passing through" all the previous layers

        ### End of your code ###

        x = self.avgpool(x)
        x = torch.flatten(x, 1)

        # Complete code here such that x is the output after the linear3 layer aft
er
        # "passing through" all the previous layers

        x = self.dropout1(x)
        x = self.linear1(x)

        x = self.relu6(x)
        x = self.dropout2(x)
        x = self.linear2(x)

        x = self.relu7(x)
        x = self.linear3(x)



        ### End of your code ###
        return x
```

Fine-tuning the Deep Neural Network

If we train such a deep network from scratch on just 50-100 images, it will not be able to learn anything meaningful, and if we had a lot more data, it would take a very long time to train (especially if you don't have a good GPU). But we don't need to! There exist pretrained models which we can train just a little more (with our 100 images); this is called *fine-tuning*.

Let's apply the pretrained weights to the model you've written (it might take ~20s the first time).

> If this step is successfull then your written model is correct!

In [6]:
```
model = DeepDeepNet()
```

In [38]:
```
from checker import check_and_apply_weights
nvidia_weights = torch.load('best_model_nvidia.pth')
check_and_apply_weights(model, nvidia_weights)
```

```
Your network seems correct!
```

Now, we'll use our collected images to (slightly) improve the pre-trained model.

Importing the necessary modules:

In [7]:
```
import torch.optim as optim
import torchvision
import torchvision.transforms as transforms
import torchvision.datasets as datasets
```

Create dataset instance

Now we use the `ImageFolder` dataset class available with the `torchvision.datasets` package. This helps us work with datasets organized into folders. We attach transforms from the `torchvision.transforms` package to prepare the data for training.

In [6]:
```
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])
    ])
)
```

Split dataset into train and test sets

Next, we split the dataset into *training* and *test* sets. A training set is the data used to (surprise, surprise) *train* the model. The test set will be used to verify the accuracy of the model we train. We never use the test set for training/improving the model.

In [7]:
```
train_size = int(len(dataset) * 0.7)
test_size = len(dataset) - train_size
train_dataset, test_dataset = torch.utils.data.random_split(dataset, [train_size,
test_size])
```

Create data loaders to load data in batches

We'll create two `DataLoader` instances, which provide utilities for shuffling data, producing *batches* of images, and loading the samples in parallel with multiple workers (processes).

In [8]:
```
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
)
```

Train the neural network (finally!)

Using the code below we will train the neural network for 20 epochs, saving the best performing model after each epoch.

> An epoch is one pass through all the training data.

In [8]:
```
device = 'cuda'
model = model.to(device)
```

> This will take 10-20 minutes. Go complete a chore or something!

In [10]:
```
NUM_EPOCHS = 20
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('Epoch %d: Accuracy = %f' % (epoch + 1, test_accuracy))
    if test_accuracy > best_accuracy:
        torch.save(model.state_dict(), BEST_MODEL_PATH)
        best_accuracy = test_accuracy
```

```
Epoch 1: Accuracy = 0.500000
Epoch 2: Accuracy = 0.500000
Epoch 3: Accuracy = 0.500000
Epoch 4: Accuracy = 0.500000
Epoch 5: Accuracy = 0.500000
Epoch 6: Accuracy = 0.500000
Epoch 7: Accuracy = 0.500000
Epoch 8: Accuracy = 0.750000
Epoch 9: Accuracy = 0.823529
Epoch 10: Accuracy = 1.000000
Epoch 11: Accuracy = 1.000000
Epoch 12: Accuracy = 0.955882
Epoch 13: Accuracy = 0.985294
Epoch 14: Accuracy = 1.000000
Epoch 15: Accuracy = 0.970588
Epoch 16: Accuracy = 1.000000
Epoch 17: Accuracy = 0.955882
Epoch 18: Accuracy = 1.000000
Epoch 19: Accuracy = 1.000000
Epoch 20: Accuracy = 1.000000
```

Trying out Collision Avoidance

Let's first load the best model:

In [9]:
```
model = DeepDeepNet()
model.load_state_dict(torch.load('best_model.pth'))
device = torch.device('cuda')
model = model.to(device)
```

Create the preprocessing 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*.

> Again, don't worry if this doesn't make sense to you.

In [10]:
```
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
```

Great! 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. We'll also create a slider that will display the probability that the robot is blocked.

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

In [12]:
```
camera = Camera.instance(width=224, height=224)
print('Camera instance created.')
image = widgets.Image(format='jpeg', width=224, height=224)
print('Image instance created.')
blocked_slider = widgets.FloatSlider(description='blocked', min=0.0, max=1.0, orie
ntation='vertical')

camera_link = traitlets.dlink((camera, 'value'), (image, 'value'), transform=bgr8_
to_jpeg)

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

```
Camera instance created.
Image instance created.
```

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

In [18]:
```
from jetbot import Robot

robot = Robot()
```

In [19]:
```
import torch.nn.functional as F
import time
```

Collision avoidance logic

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. While the neural network output indicates we're blocked, we'll turn left, otherwise we go forward.

In the code block below, `prob_block` is the output of the neural network which is a number in $[0, 1]$. Complete the code below such that the robot moves forward if `prob_block` < 0.5 with a speed of `0.20` . Otherwise, it moves its left wheel with a speed of `0.20` (spinning in place).

In [23]:
```
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 distribution)
    y = F.softmax(y, dim=1)

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

    blocked_slider.value = prob_blocked

    ### Write your code here ###

    if prob_blocked < .5:
        robot.left_motor.value=0.25
        robot.right_motor.value=0.23
    else:
        robot.left_motor.value=0.0


    ### End of your code ###


    time.sleep(0.001)

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

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. The collision avoidance should work, but the neural network is only as good as the data it's trained on!

In [24]:
```
camera.observe(update, names='value')  # this attaches the 'update' function to th
e 'value' traitlet of our camera
```

Awesome! If your robot is plugged in it should now be generating new commands with each new camera frame. Perhaps start by placing your robot on the ground and seeing what it does when it reaches an obstacle.

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

In [25]:
```
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.

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

To continue streaming call the following.

In [ ]:
```
camera_link.link()  # stream to browser (wont run camera)
```

Conclusion

That's it for this live demo! Hopefully you had some fun and your robot avoided collisions intelligently!

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

https://www.youtube.com/watch?v=ttQ7px6FPic&feature=youtu.be (https://www.youtube.com/watch?v=ttQ7px6FPic&feature=youtu.be)

In [ ]:
```
```