LAMPIRAN

1. Jetbot *Assembly Process*

2. *Samples Dataset*

3. *Coding (Dataset Collecting)*

Display live camera feed

Initialize and display camera

> Neural network takes a 224x224 pixel image as input. Set the camera to that size to minimize the filesize of dataset.

```
[1]: 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 to match the camera

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

     display(image)
```

Create Directory for dataset

```
[ ]: import os

     blocked_left_dir = 'dataset/blocked_left'
     blocked_right_dir = 'dataset/blocked_right'
     free_dir = 'dataset/free'

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

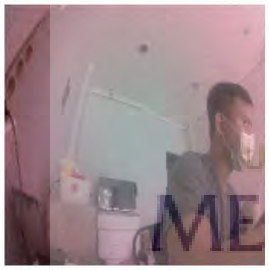

Create UI for Camera capturing process

```
[3]: button_layout = widgets.Layout(width='128px', height='64px')
     free_button = widgets.Button(description='add free', button_style='success', layout=button_layout)
     blocked_left_button = widgets.Button(description='add blocked_left', button_style='danger', layout=button_layout)
     blocked_right_button = widgets.Button(description='add blocked_right', button_style='danger', layout=button_layout)
     free_count = widgets.IntText(layout=button_layout, value=len(os.listdir(free_dir)))
     blocked_left_count = widgets.IntText(layout=button_layout, value=len(os.listdir(blocked_left_dir)))
     blocked_right_count = widgets.IntText(layout=button_layout, value=len(os.listdir(blocked_right_dir)))

     display(widgets.HBox([free_count, free_button]))
     display(widgets.HBox([blocked_left_count, blocked_left_button]))
     display(widgets.HBox([blocked_right_count, blocked_right_button]))
```

Create Save snapshot to directory function

[4]:
```
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_left():
    global blocked_left_dir, blocked_left_count
    save_snapshot(blocked_left_dir)
    blocked_left_count.value = len(os.listdir(blocked_left_dir))

def save_blocked_right():
    global blocked_right_dir, blocked_right_count
    save_snapshot(blocked_right_dir)
    blocked_right_count.value = len(os.listdir(blocked_right_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_left_button.on_click(lambda x: save_blocked_left())
blocked_right_button.on_click(lambda x: save_blocked_right())
```

[5]:
```
display(image)
display(widgets.HBox([free_count, free_button]))
display(widgets.HBox([blocked_left_count, blocked_left_button]))
display(widgets.HBox([blocked_right_count, blocked_right_button]))
```



30 add free

75 add blocked_left

75 add blocked_right

4. *Coding (Neural Networks Training)*

Collision Avoidance - Train Model

[1]:
```
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
```

Create dataset instance

[2]:
```
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

Split the dataset into *training* and *test* sets. The test set will be used to verify the accuracy of the model we train.

[3]:
```
train_dataset, test_dataset = torch.utils.data.random_split(dataset, [len(dataset) - 50, 50])
```

Create data loaders to load data in batches ¶

Create two `DataLoader` instances, which provide utilities for shuffling data, producing *batches* of images, and loading the samples in parallel with multiple workers.

[4]:
```
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
)
```


Define the neural network

torchvision package pre-trained models.

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

[5]:
```
model = models.alexnet(pretrained=True)
```

#Change AlexNet final layer into 3 label classifier

[6]:
```
model.classifier[6] = torch.nn.Linear(model.classifier[6].in_features, 3)
```

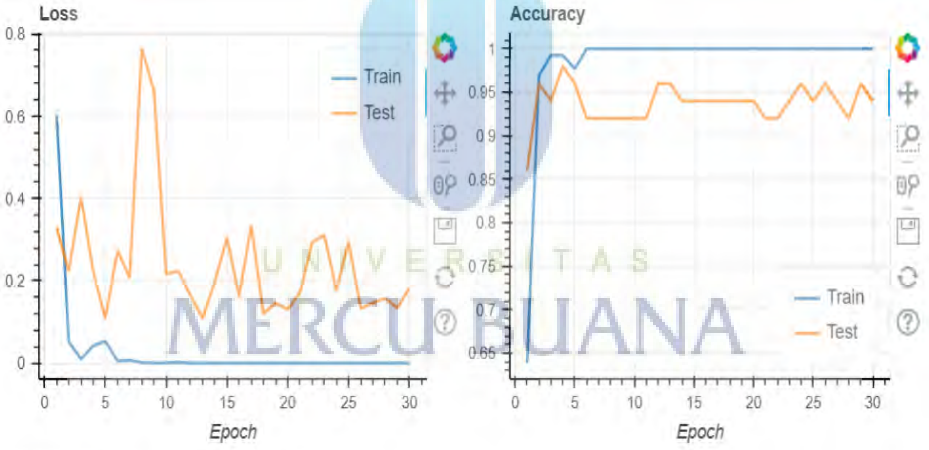
Transfer the model for execution on the GPU

[7]:
```
device = torch.device('cuda')
model = model.to(device)
```

Train the neural network

> An epoch is a full run through our data.

[8]:
```
from bokeh.io import push_notebook, show, output_notebook
from bokeh.layouts import row
from bokeh.plotting import figure
from bokeh.models import ColumnDataSource
from bokeh.models.tickers import SingleIntervalTicker
output_notebook()

colors = ['#1f77b4', '#ff7f0e', '#2ca02c', '#d62728']

p1 = figure(title="Loss", x_axis_label="Epoch", plot_height=300, plot_width=360)
p2 = figure(title="Accuracy", x_axis_label="Epoch", plot_height=300, plot_width=360)

source1 = ColumnDataSource(data={'epochs': [], 'trainlosses': [], 'testlosses': [] })
source2 = ColumnDataSource(data={'epochs': [], 'train_accuracies': [], 'test_accuracies': []})

#r = p1.multi_line(ys=['trainlosses', 'testlosses'], xs='epochs', color=colors, alpha=0.8, legend_label=['Training','Test'], source=sour
r1 = p1.line(x='epochs', y='trainlosses', line_width=2, color=colors[0], alpha=0.8, legend_label="Train", source=source1)
r2 = p1.line(x='epochs', y='testlosses', line_width=2, color=colors[1], alpha=0.8, legend_label="Test", source=source1)

r3 = p2.line(x='epochs', y='train_accuracies', line_width=2, color=colors[0], alpha=0.8, legend_label="Train", source=source2)
r4 = p2.line(x='epochs', y='test_accuracies', line_width=2, color=colors[1], alpha=0.8, legend_label="Test", source=source2)

p1.legend.location = "top_right"
p1.legend.click_policy="hide"
```




```
1: 0.602306, 0.328046, 0.638462, 0.860000
2: 0.052278, 0.224863, 0.969231, 0.960000
3: 0.009738, 0.404261, 0.992308, 0.940000
4: 0.042524, 0.223551, 0.992308, 0.980000
5: 0.053239, 0.108321, 0.976923, 0.960000
6: 0.005829, 0.274450, 1.000000, 0.920000
```

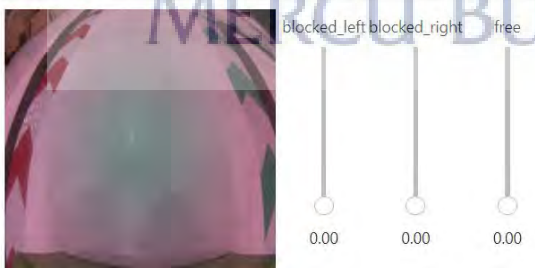
5. *Coding (ROS Operation)*

Create the preprocessing function

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

[4]:
```
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
```

```
camera = Camera.instance(width=224, height=224)
image = widgets.Image(format='jpeg', width=224, height=224)
blocked_left_slider = widgets.FloatSlider(description='blocked_left', min=0.0, max=1.0, orientation='vertical')
blocked_right_slider = widgets.FloatSlider(description='blocked_right', min=0.0, max=1.0, orientation='vertical')
free_slider = widgets.FloatSlider(description='free', 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_left_slider, blocked_right_slider, free_slider]))
```



```
[ ] from jetbot import Robot

    robot = Robot()
```

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

    def update(change):
        global blocked_left_slider,blocked_right_slider,free_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_left = float(y.flatten()[0])
        prob_blocked_right = float(y.flatten()[1])
        prob_free = float(y.flatten()[2])
        blocked_left_slider.value = prob_blocked_left
        blocked_right_slider.value = prob_blocked_right
        free_slider.value = prob_free

        if prob_blocked_left > 0.5:
            robot.right(0.1)
        elif prob_blocked_right > 0.5:
            robot.left(0.1)
        else:
            robot.forward(0.125)

        time.sleep(0.001)
```


```
[6]: from jetbot import Robot

     robot = Robot()
```

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

     def update(change):
         global blocked_left_slider,blocked_right_slider,free_slider, robot
         x = change['new']
```

6. *LINK* SITASI (GDRIVE)

TYPE	LINK
E-Book	https://drive.google.com/drive/folders/1Yq_G1daUxLvOY7NuaG0fn_gN6_oXl7GE?usp=sharing
Journals	https://drive.google.com/drive/folders/1MCWltVMsbWUttNW7zuX2JobR54URM6lI?usp=sharing