live_einsatz_kollisionsvermeidung

September 25, 2022

1 Kollisionsvermeidung - Live Demo

In diesem Notebook wird das trainierte Model eingesetzt um festzustellen, ob der Roboter `free` oder `blocked` ist, um die Fähigkeit der Kollisionsvermeidung zu implementieren.

1.1 Laden des trainierten Modells

Die Datei `best_model.pth` muss nun wieder auf den Jetbot geladen und in den Ordner dieses Notebooks kopiert werden.

Der folgende Codeblock initialisiert das PyTorch Modell. Es fällt die Ähnlichkeit zum Training auf.

[ ]:
```
import torch
import torchvision

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

Als nächstes werden die trainierten Weighs aus der hochgeladenen `best_model.pth` Datei geladen.

[ ]:
```
model.load_state_dict(torch.load('best_model.pth'))
```

Aktuell befinden sich diese noch auf dem Speicher der CPU. Um sie auf die GPU zu übertragen, muss der folgende Codeblock ausgeführt werden.

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

1.1.1 Erstellen der Vorverarbeitungs-Funktion (Proprocessing)

Das Modell wurde nun geladen, jedoch existiert noch ein kleines Problem. Das Format des trainierten modells stimmt nicht *exakt* mit dem der Kamera überein. Um das zu korrigieren sind die folgenden *preprocessing* Bildverarbeitsungsschritte nötig:

1. Konvertieren von BGR zu RGB
2. Konvertieren von HWC Layout zum CHW Layout
3. Normalisieren unter der Nutzung der selben Parameter wie im Training (die Kamera gibt Werte zwischen [0, 255] zurück, die geladenen Bilder haben jedoch einen Wertebereich von [0, 1]. Folglich muss um 255.0 skaliert werden)
4. Transferieren der Daten vom CPU Speicher (RAM) zum GPU Speicher (VRAM)

5. Paketgrößen hinzufügen (Batchgröße)

[ ]:
```
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
```

Nun wurde die preprocessing-Funktion erstellt, die die oben genannten Schritte ausführt, um das Kamera-Format an das des trainierten Modells anzupassen.

Im nächsten Schritt soll das Kamerabild wieder angezeigt werden. Weiterhin wird ein Schieberegler erstellt, der anzeigen soll, wie hoch die Wahrscheinlichkeit ist, dass der Roboter `blocked` ist. Außerdem wirds ein Regler implementiert, über den die Geschwindigkeit des JEtbot eingestellt werden kann.

[ ]:
```
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')
speed_slider = widgets.FloatSlider(description='speed', min=0.0, max=0.5,␣
 ↪value=0.0, step=0.01, orientation='horizontal')

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

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

Weiterhin muss wieder die `robot` Instanz erstellt werden, die für die Steuerung der Motoren benötigt wird.

[ ]:
```
from jetbot import Robot

robot = Robot()
```

Darauf wird die Funktion implementiert, die immer dann aufgerufen wird, wenn eine Änderung des Kamerawertes (Kamerabildes) stattfindet. Diese Funktion schließt folgende Schritte ein:

1. Pre-process des Kamerabildes
2. Aufrufen/Ausführen des neuronalen Netzes
3. Wenn das neuronale netzt `blocked` als Ergebnis liefert, dann soll der Roboter sich nach links drehen, ansonsten soll er geradeaus fahren.

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

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

    blocked_slider.value = prob_blocked

    if prob_blocked < 0.5:
        robot.forward(speed_slider.value)
    else:
        robot.left(speed_slider.value)

    time.sleep(0.001)

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

Nachdem die Ausführ-Funktion erstellt wurde, muss diese nun an die Kamera verknüpft werden, um die Verarbeitung zu ermöglichen.

Dies kann über die `observe` Funktion erreicht werden.

ACHTUNG: der Jetbot wird sich nun von alleine bewegen!

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

Soll der Roboter wieder gestoppt werden, kann dies über die `unobserve` Funktion erreicht werden.

[ ]:
```
import time

camera.unobserve(update, names='value')

time.sleep(0.1)  # add a small sleep to make sure frames have finished␣
 ↪processing

robot.stop()
```

Falls das Kamerabild nicht dauerhaft im Browser angezeigt werden soll:

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

Um das Kamerabild wieder im Browser anzuzeigen:

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

Am Ende der Ausführung sollte die Kameraverbindung wieder getrennt werden, damit diese erneut in einem anderen Notebook verwendet werden kann.

[ ]:
```
camera.stop()
```