live_einsatz_spurverfolgung

September 25, 2022

1 Spurverfolgung - Live Demo

In diesem Notebook wird das trainierte Modell genutzt um den Jetbot auf einer (Klemmbaustein-) Straße fahren zu lassen.

1.0.1 Laden des trainierten Modells

Nun muss die `best_steering_model_xy.pth` Datei wieder auf den Jetbor hochgeladen werden in den Ordner dieses Notebooks.

Der folgende Code initialisiert das PyTorch Modell.

[ ]:
```
import torchvision
import torch

model = torchvision.models.resnet18(pretrained=False)
model.fc = torch.nn.Linear(512, 2)
```

Als nächstes werden die trainierten Weights aus der `best_steering_model_xy.pth` Datei hochgeladen.

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

Aktuell sind die hochgeladenen Weights noch im Speicher der CPU hinterlegt. Diese müssen zunächst in den VRAM der GPU transferiert werden.

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

1.0.2 Erstellen der Proprocessing Funktion

Wie bereits bei der Kollisionsvermeidung muss auch hier ein Proprocessing der Bilddaten stattfinden. Dazu werden die folgenden Schritte ausgeführt:

1. Kovertieren vom HWC Layout zum CHW Layout
2. 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)
3. Transferieren der Daten vom CPU Speicher (RAM) zum GPU Speicher (VRAM)

4. Paketgrößen hinzufügen (Batchgröße)

```
[ ]: import torchvision.transforms as transforms
     import torch.nn.functional as F
     import cv2
     import PIL.Image
     import numpy as np

     mean = torch.Tensor([0.485, 0.456, 0.406]).cuda().half()
     std = torch.Tensor([0.229, 0.224, 0.225]).cuda().half()

     def preprocess(image):
         image = PIL.Image.fromarray(image)
         image = transforms.functional.to_tensor(image).to(device).half()
         image.sub_(mean[:, None, None]).div_(std[:, None, None])
         return image[None, ...]
```

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.

```
[ ]: from IPython.display import display
     import ipywidgets
     import traitlets
     from jetbot import Camera, bgr8_to_jpeg

     camera = Camera()

     image_widget = ipywidgets.Image()

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

     display(image_widget)
```

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

```
[ ]: from jetbot import Robot

     robot = Robot()
```

Als nächstes werden Schieberegler, über die der Roboter konfiguriert werden kann, implementiert. > Anmerkung: Die Regler wurden nach bestem Wissen und Wissen vorkonfiguriert. Konkrete Einstellungen hänger in der Praxis jedoch stark von der Umgebung und den TRainingsdaten ab.

1. Geschwindigkeitseinstellung (speed_gain_slider): Damit der Jetbot losfährt muss der `speed_gain_slider` erhöht werden

2. Lenk-Verstärkung (steering_gain_slider): Falls der Jetbot hin- und her wackeln sollte muss der `steering_gain_slider` verringert werden, bis dieser sich flüssig fortbewegt
3. Vorsteuerung der Lenkung (steering_bias_slider): Sollte sich der Jetbot permanent zu sehr in eine Richtung drehen, kann der `steering_bias_slider` verwendet werden um die Lenkung zu korrigieren (das gilt sowohl für eine Korrektur der Motoren als auch der Kameraposition)

   Anmerkung: Es ist hilfreich bei einer niedrigen Geschwindigkeit mit den Reglern "rumzuspielen" um die optimalen Einstellungen zu finden.

[ ]:
```
speed_gain_slider = ipywidgets.FloatSlider(min=0.0, max=1.0, step=0.01,␣
 ↪description='speed gain')
steering_gain_slider = ipywidgets.FloatSlider(min=0.0, max=1.0, step=0.01,␣
 ↪value=0.2, description='steering gain')
steering_dgain_slider = ipywidgets.FloatSlider(min=0.0, max=0.5, step=0.001,␣
 ↪value=0.0, description='steering kd')
steering_bias_slider = ipywidgets.FloatSlider(min=-0.3, max=0.3, step=0.01,␣
 ↪value=0.0, description='steering bias')

display(speed_gain_slider, steering_gain_slider, steering_dgain_slider,␣
 ↪steering_bias_slider)
```

Als nächstes werden Schieberegler angelegt, die die Vorhersagen des Modells bezüglich des x- und y-Wertes anzeigen sollen.

Der Link-Schieberegler zeigt den geschätzten Lenkwert an.

[ ]:
```
x_slider = ipywidgets.FloatSlider(min=-1.0, max=1.0, description='x')
y_slider = ipywidgets.FloatSlider(min=0, max=1.0, orientation='vertical',␣
 ↪description='y')
steering_slider = ipywidgets.FloatSlider(min=-1.0, max=1.0,␣
 ↪description='steering')
speed_slider = ipywidgets.FloatSlider(min=0, max=1.0, orientation='vertical',␣
 ↪description='speed')

display(ipywidgets.HBox([y_slider, speed_slider]))
display(x_slider, steering_slider)
```

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. Berechnung der ungefähren Lenkwerte
4. Steuerung der Motoren über PD-Regler

[ ]:
```
angle = 0.0
angle_last = 0.0

def execute(change):
```

```
    global angle, angle_last
    image = change['new']
    xy = model(preprocess(image)).detach().float().cpu().numpy().flatten()
    x = xy[0]
    y = (0.5 - xy[1]) / 2.0

    x_slider.value = x
    y_slider.value = y

    speed_slider.value = speed_gain_slider.value

    angle = np.arctan2(x, y)
    pid = angle * steering_gain_slider.value + (angle - angle_last) *␣
 ↪steering_dgain_slider.value
    angle_last = angle

    steering_slider.value = pid + steering_bias_slider.value

    robot.left_motor.value = max(min(speed_slider.value + steering_slider.
 ↪value, 1.0), 0.0)
    robot.right_motor.value = max(min(speed_slider.value - steering_slider.
 ↪value, 1.0), 0.0)

execute({'new': camera.value})
```

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(execute, names='value')
```

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

[ ]:
```
import time

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

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

robot.stop()
```

Als letztes wird wieder die Verbindung zur Kamera getrennt

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