datensammlung_kollisionsvermeidung

September 25, 2022

1 Kollisionsvermeidung - Datensammlung

Im nächsten Schritt soll es darum gehen den Jetbot von selbst fahren zu lassen.

Dabei handelt es sich um eine schwierige Aufgabe, die jedoch in kleinere Teilaufgaben unterteilt werden kann. Eine der wichtigsten Aufgaben bzw. Fähigkeiten, die der Roboter besitzen muss ist, dass er sich nicht selbst in gefährliche Situationen begeben kann. Diese Fähigkeit wird nachfolgend als *Kollisionsvermeidung* bezeichnet.

Die folgenden Notebooks beschäftigen sich mit der Umsetzung dieser Fähigkeit mithilfe von Neuronalen Netzen und der Kamera des Roboters als Sensor.

Der Ansatz ist dabei eine virtualle "safety bubble" um den Roboter herum zu erstellen. Innerhalb dieser Blase kann dieser sich frei im Kreis drehen, ohne dass er mit Hindernissen kollidiert oder von einem Vorsprung herunterfällt.

Konkret würd dafür wie folgt vorgegangen:

Als erstes wird der Jetbot in Szenarien platziert, wo seine "safety bubble" verletzt wird. Dazu werden Bilder aufgenommen zusammen mit dem Label `blocked`.

Als zweites wird er in Szenarien platziert, wo er sich nach vorne bewegen könnte, ohne dass er mit etwas Kollidiert. Es werden ebenfalls Bilder aufgenommen, diesmal jedoch mit dem Label `free`.

Nachdem genügend gelabelte Bilder aufgenommen wurden (ca. 100 sollten fürs erste genügen), werden diese auf einen Computer mit einer schnellen GPU geladen, um dort das neuronale Netzwerk zu *trainieren*, so dass es anhand der aufgenommenen Bilder entscheiden kann, ob die safety bubble des Jetbots verletzt wurde. Damit wird es möglich sein am Ende die Fähigkeit der Kollisionsvermeidung zu implementieren.

1.0.1 Live-Bild der Kamera anzeigen

Zunächst wird die Kamera initialisiert und ihr Bild angezeigt.

> Das neuronale Netzwerk erwartet Bilder mit der Auflösung von 224x224 Pixeln als Eingabe. Dafür wird die Auflösung der Kamera entsprechend dieser Anforderung gesetzt, was ebenfalls dafür sorgt, dass die Dateigröße minimiert wird. Es gibt grundsätzlich auch Szenarien, wo es von Vorteil wäre die volle Auflösung zu nutzen und erst zu einen späteren Zeitpunkt die Bilder zu komprimieren. Dies wird hier jedoch nicht behandelt.

[ ]:
```
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)
```

Als nächsten werden Verzeichnisse angelegt, in denen die Bild-Daten gespeichert werden. Es wird ein Verzeichnis `dataset` angelegt, welches zwei Unterordner `free` und `blocked` enthält. In diesen werden die Bilder für die jeweiligen Szenarien gespeichert.

[ ]:
```
import os

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

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

Als nächstest werden Buttons angelegt, mit denen die Bilder mitsamt Label gespeichert werden können. Außerdem werden Textfelder angelegt, die anzeigen, wie viele Bilder für die jeweiligen Kategorien bereits gespeichert wurden. Dies ist hilfreich, um sicherzustellen, dass etwa gleich viele Bilder für die Kategorien `free` und `blocked` gespeichert wurden. Außerdem hilft es zu wissen, wie viele Bilder insgesamt gespeichert wurden.

[ ]:
```
button_layout = widgets.Layout(width='128px', height='64px')
free_button = widgets.Button(description='add free', button_style='success',␣
 ↪layout=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_dir)))
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]))
```

Aktuell haben die Buttons noch keine Funktion. Dafür müssen die Funktionen zum speichern der Bilder der jeweiligen Kategorie erst mit dem `on_click` Event der Buttons verbunden werden. Die Bilder werden dabei aus dem `Image` Widget gespeichert, da diese bereits im komprimierten JPEG Format vorliegen.

Um sicherzugehen, dass keine Dateinamen doppelt vergeben werden (auch nicht auf verschiedenen Computern), wird das `uuid` Paket in Python verwendet, welches die `uuid1` Methode definiert, um einen eindeutigen Identifikator zu generieren. Dieser eindeutige Identifikator wird aus Informationen wie der aktuellen Zeit und der Maschinenadresse generiert.

[ ]:
```
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())
```

Nun können die Buttons Bilder in den `free` und `blocked` Ordnern abspeichern.

Als nächstes kann mit dem Aufnehmen von Bildern begonnen werden:

1. Jetbot in einem Szenario platzieren, wo er blockiert ist und `add blocked` drücken
2. Roboter in einem Szenario platzieren, wo er frei ist und den Button `add free` betätigen
3. 1 und 2 wiederholen, bis genügend Bilder gespeichert wurden

Tipps für das Labeln der Daten:

1. Verschiedene Ausrichtungen des Roboters aufnehmen
2. Verschiedene Lichtverhältnisse aufnehmen
3. Verschiedene Typen von Kollisionen aufnehmen (z.B. Wände, Vorsprünge, Gegenstände)

4. Verschiedene Texturen aufnehmen (z.B. glatte Oberflächen, rauhe Oberflächen, etc.)

Grundsätzlich kann gesagt werden, um so mehr Daten aufgenommen wurden für verschiedene Szenarien, desto besser ist das kollisionsvermeidende Verhalten des Roboters. Dabei ist es vor allem wichtig eine *Varianz* in den Daten zu haben (wie oben beschrieben) und nicht nur eine große Menge an Bildern.

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

Abschließend muss die Kameraverbindung wieder geschlossen werden, damit diese später in einem anderen NOtebook wieder genutzt werden kann.

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

1.1 Wie geht es weiter?

Wurden genügend DAten gesammelt, können diese auf einen Computer mit einer Leistungsstarken GPU kopiert werden. Dafür kann das folgende *Terminal* Kommando aufgerufen werden, um dataset Ordner in eine *zip*-Datei umzuwandeln.

> Der ! Prefix gibt an, dass die Zelle als *shell* (oder *terminal*) Kommando ausgeführt werden soll.
>
> Die -r Flag im zip Kommando indiziert *recursive*, dass alle untergeordneten Dateien mit berücksichtigt werden, die -q Flag indiziert *quiet*, damit das zip Kommando keine Ausgabe in das Terminal auslöst.

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

Die `dataset.zip` Datei kann nun heruntergeladen und auf den Zielrechner kopiert werden. You should download the zip file using the Jupyter Lab file browser by right clicking and selecting `Download`.

Im nächsten Schritt und gleichzeitig auch nächsten Notebook wird dann das neuronale Netz trainiert.