modell_training_spurverfolgung

September 25, 2022

1 Spurverfolgung - Modell-Training

In diesem Notebook wird ein neuronales Netzwerk trainiert ein Eingabebild entgegenzunehmen und eine Menge von x, y-Werten wieder auszugeben, die einer Zielposition (des Roboters) entsprechen.

Dafür wird das PyTorch Deep Learning Framework verwendet, um ein ResNet18-Modell für die Spurverfolgung zu trainieren.

[ ]:
```
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
import glob
import PIL.Image
import os
import numpy as np
```

1.0.1 Downloaden und Extrahieren des Datensatzes

Die `road_following_<Date&Time>.zip` muss zunächst entpackt werden.

Dazu dient folgender Befehl:

[ ]:
```
!unzip -q road_following.zip
```

Es sollte nun ein Ordner mit dem Namen `dataset_all` erstellt worden sein.

1.0.2 Datensatz-Instanz erstellen

Hier wird eine eigene `torch.utils.data.Dataset` Implementation erstellt, welche die `__len__` und `__getitem__` Funktionen implementiert. Diese KLasse ist dafür verantwortlich Bilder zu laden und die x, y-Werte aus den Bilddateinamen zu parsen. Da die KLasse `torch.utils.data.Dataset` implementiert wurde, können alle torch-Daten-Utilities verwendet werden.

Außerdem wurden einige Transformationen hard gecoded (wie z.B. ein Farb-Jitter) des Datensatzes. Außerdem wurden zufällige horizontale Spiegelungen der Bilder durchgeführt, falls es sich bei der

Strecke um einen nicht-symmetrischen Pfad handelt (z.B. eine Straße, auf der der Roboter auf der rechten Seite fahren muss).

[ ]:
```
def get_x(path, width):
    """Gets the x value from the image filename"""
    return (float(int(path.split("_")[1])) - width/2) / (width/2)

def get_y(path, height):
    """Gets the y value from the image filename"""
    return (float(int(path.split("_")[2])) - height/2) / (height/2)

class XYDataset(torch.utils.data.Dataset):

    def __init__(self, directory, random_hflips=False):
        self.directory = directory
        self.random_hflips = random_hflips
        self.image_paths = glob.glob(os.path.join(self.directory, '*.jpg'))
        self.color_jitter = transforms.ColorJitter(0.3, 0.3, 0.3, 0.3)

    def __len__(self):
        return len(self.image_paths)

    def __getitem__(self, idx):
        image_path = self.image_paths[idx]

        image = PIL.Image.open(image_path)
        width, height = image.size
        x = float(get_x(os.path.basename(image_path), width))
        y = float(get_y(os.path.basename(image_path), height))

        if float(np.random.rand(1)) > 0.5:
            image = transforms.functional.hflip(image)
            x = -x

        image = self.color_jitter(image)
        image = transforms.functional.resize(image, (224, 224))
        image = transforms.functional.to_tensor(image)
        image = image.numpy()[::-1].copy()
        image = torch.from_numpy(image)
        image = transforms.functional.normalize(image, [0.485, 0.456, 0.406],␣
 ↪[0.229, 0.224, 0.225])

        return image, torch.tensor([x, y]).float()

dataset = XYDataset('dataset_xy', random_hflips=False)
```

1.0.3 Aufteilen des Datensatzes in Trainings- und Validierungsdatensatz

Nachdem der Datensatz eingelesen wurde wird er auch hier in einen Trainings- und Validierungsdatensatz aufgeteilt. In diesem Beispiel wird der Datensatz in 90%-10% aufgeteilt. Der Validierungsdatensatz wird verwendet um die Genauigkeit des trainierten Modells zu überprüfen.

[ ]:
```
test_percent = 0.1
num_test = int(test_percent * len(dataset))
train_dataset, test_dataset = torch.utils.data.random_split(dataset,␣
 ↪[len(dataset) - num_test, num_test])
```

1.0.4 Erstellen von DataLoader-Instanzen zum Laden von Daten in Batches

Es wird ein `DataLoader` erstellt, um die Daten in Batches zu laden, diese zu mischen und das Laden in mehreren (Unter-) Prozessen zu ermöglichen. In diesem Beispiel wird eine Batch-Größe von 64 verwendet. Die Batch-Größe sollte auf die verfügbare GPU-Speichergröße abgestimmt sein, da sie die Genauigkeit des Modells beeinflusst.

[ ]:
```
train_loader = torch.utils.data.DataLoader(
    train_dataset,
    batch_size=8,
    shuffle=True,
    num_workers=0
)

test_loader = torch.utils.data.DataLoader(
    test_dataset,
    batch_size=8,
    shuffle=True,
    num_workers=0
)
```

1.0.5 Deklarieren des neuronalen Netzes

Es wird das ResNet-18 Modell aus PyTorch TorchVision verwendet.

Auch hier findet das Transfer Learning wieder Anwendung.

[ ]:
```
model = models.resnet18(pretrained=True)
```

Das ResNet Modell hat eine fully connected (fc) final-Layer mit 512 `in_features` und nach dem Angewendeten Training (unter Nutzung von Regression) mit einem `out_features`.

Abschließend wir das Modell zur Berechnung auf die GPU verschoben.

[ ]:
```
model.fc = torch.nn.Linear(512, 2)
device = torch.device('cuda')
model = model.to(device)
```

1.0.6 Regressionstraining

Es wird für 50 Epochen trainiert und das beste Modell wird abgespeichert, falls der Verlust im vergleich zur vorherigen Epoche verringert werden konnte.

[ ]:
```
NUM_EPOCHS = 70
BEST_MODEL_PATH = 'best_steering_model_xy.pth'
best_loss = 1e9

optimizer = optim.Adam(model.parameters())

for epoch in range(NUM_EPOCHS):

    model.train()
    train_loss = 0.0
    for images, labels in iter(train_loader):
        images = images.to(device)
        labels = labels.to(device)
        optimizer.zero_grad()
        outputs = model(images)
        loss = F.mse_loss(outputs, labels)
        train_loss += float(loss)
        loss.backward()
        optimizer.step()
    train_loss /= len(train_loader)

    model.eval()
    test_loss = 0.0
    for images, labels in iter(test_loader):
        images = images.to(device)
        labels = labels.to(device)
        outputs = model(images)
        loss = F.mse_loss(outputs, labels)
        test_loss += float(loss)
    test_loss /= len(test_loader)

    print('%f, %f' % (train_loss, test_loss))
    if test_loss < best_loss:
        torch.save(model.state_dict(), BEST_MODEL_PATH)
        best_loss = test_loss
```

Ist das Modell fertig trainiert, so wird es in der `best_steering_model_xy.pth` Datei abgelegt, welche wieder zurück auf den Jetbot geladen werden kann.