modell_training_kollisionsvermeidung

September 25, 2022

1 Kollisionsvermeidung - Modell-Training

In diesem Notebook wird ein Bildklassifikator darauf trainiert die zwei Klassen `free` und `blocket` zur Vermeidung von Kollisionen zu erkennen. Dazu wird die Deep-Learning-Bibliothek *PyTorch* verwendet.

[ ]:
```
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
```

1.0.1 Datensatz hochladen und entpacken

Bevor begonnen werden kann wird die `dataset.zip` Datei aus dem `datensammlung.ipynb` Notebook vom Roboter hochgeladen.

Der folgende Befehl entpackt die Zip-Datei:

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

Es sollte nun ein `dataset` Ordner erscheinen.

1.0.2 Datensatz-Instanz erstellen

Als nächstes wird die `ImageFolder` Datensatz-Klasse aus dem `torchvision.datasets` Paket genutzt. Weiterhin werden sogenannte transform aus dem `torchvision.transforms` Paket verwendet, um die Daten für das Trainieren vorzubereiten.

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

```
)
```

1.0.3 Aufteilen des Datensatzes in Trainings- und Testdaten

Als nächstes wird der Datensatz in *training* und *test* Sets unterteilt. Die Test-Sets werden genutzt um die Genauigkeit/Performance des antrainierten Modells zu verifizieren.

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

1.0.4 Data Loaders zum Laden der Daten in in Paketen

Es werden zwei `DataLoader` Instanzen erzeugt, welche Fähigkeiten zum durchmischen von Daten bereitstellen, sowie *batches* von Bildern erzeugen, die dann parallel an das Modell übergeben werden können mittels meherere Worker.

[ ]:
```
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 Definieren des neuronalen Netzes

Nun wird das neuronale Netz deklariert, welches im Folgenden genutzt werden soll. Das *torchvision* Paket besitzt bereits eine sammlung von vortrainierten Modellen, die genutzt werden können.

Über das *Transfer Learning* kann ein vortrainiertes Modell weiterverwertet werden für eine neue Aufgabe, für weelche deutlich weniger Daten zur Verfügung stehen.

Wichtige Eigenschaften, die beim initialen Training des vorttrainierten Modells erlernt wurden, können für die neue Aufgabe weiterverwertet werden. Als neuronales Netzwerk wird anschließend das `alexnet`-Modell genutzt werden.

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

Das `alexnet`-Modell wurde ursprünglich mit einem Datensatz aus 1000 Klasen-Labeln trainiert, der nun genutze Datensatz besitz jedoch nur zwei Label. Folglich wird die letzte Layer mit einer neuen, untrainierten Layer ersetzt, welche nur zwei Ausgaben besitzt.

[ ]:
```
model.classifier[6] = torch.nn.Linear(model.classifier[6].in_features, 2)
```

Abschließend wird das Modell auf die GPU verschoben, um dort berechnet zu werden

`cuda` sind die Rechenkerne einer Nvidia GPU

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

1.0.6 Trainieren das neuronalen Netzes

Über den unter stehenden Code wird das neuronale Netz in 30 Epochen trainiert, wobei das beste Modell nach jeder Epoche gespeichert wird.

Eine Epoche ist ein Durchlauf durch den gesamten Trainingsdatensatz

[ ]:
```
NUM_EPOCHS = 30
BEST_MODEL_PATH = 'best_model.pth'
best_accuracy = 0.0

optimizer = optim.SGD(model.parameters(), lr=0.001, momentum=0.9)

for epoch in range(NUM_EPOCHS):

    for images, labels in iter(train_loader):
        images = images.to(device)
        labels = labels.to(device)
        optimizer.zero_grad()
        outputs = model(images)
        loss = F.cross_entropy(outputs, labels)
        loss.backward()
        optimizer.step()

    test_error_count = 0.0
    for images, labels in iter(test_loader):
        images = images.to(device)
        labels = labels.to(device)
        outputs = model(images)
        test_error_count += float(torch.sum(torch.abs(labels - outputs.
 ↪argmax(1))))

    test_accuracy = 1.0 - float(test_error_count) / float(len(test_dataset))
    print('%d: %f' % (epoch, test_accuracy))
    if test_accuracy > best_accuracy:
        torch.save(model.state_dict(), BEST_MODEL_PATH)
        best_accuracy = test_accuracy
```

Wurde der Vorgang komplett abgeschlossen, sollte eine `best_model.pth` Datei im Jupyter Lab Dateibrowser erscheinen, welche anschließend heruntergaladen werden kann.