aVogEphzXQ7vPWrc0SItkmnKztAm7BQwtep/mcip

AKIA5LHYJWKZEY3W4DMD

echo 'export PATH="/usr/local/bin:/opt/homebrew/bin:/usr/bin:/bin:/usr/sbin:/sbin:$PATH"' >> ~/.zshrc

source ~/.zshrc

export PATH="/Applications/Docker.app/Contents/Resources/bin:$PATH"

ghp\_kJkavN0EsSg1kFhxAIO65reVnfKYQs1a8Ixw

import os

import torch

import torch.nn as nn

import torch.nn.functional as F

from torchvision import datasets, transforms

from torch.utils.data import DataLoader

import json

# Verzeichnisse

train\_dir = "data/images/train"

val\_dir = "data/images/val"

test\_dir = "data/images/test"

# Bildvorverarbeitung (z. B. Resize, Tensor, Normalisierung)

train\_transform = transforms.Compose([

transforms.Resize((128, 128)),

transforms.RandomHorizontalFlip(),

transforms.RandomRotation(10),

transforms.RandomResizedCrop(128, scale=(0.8, 1.0)),

transforms.ToTensor(),

transforms.Normalize(mean=[0.5], std=[0.5])

])

val\_test\_transform = transforms.Compose([

transforms.Resize((128, 128)),

transforms.ToTensor(),

transforms.Normalize(mean=[0.5], std=[0.5])

])

# Datasets mit separaten Transforms

train\_dataset = datasets.ImageFolder(train\_dir, transform=train\_transform)

val\_dataset = datasets.ImageFolder(val\_dir, transform=val\_test\_transform)

test\_dataset = datasets.ImageFolder(test\_dir, transform=val\_test\_transform)

# Dataloader

train\_loader = DataLoader(train\_dataset, batch\_size=32, shuffle=True)

val\_loader = DataLoader(val\_dataset, batch\_size=32, shuffle=False)

test\_loader = DataLoader(test\_dataset, batch\_size=32, shuffle=False)

class InsuranceCNN(nn.Module):

def \_\_init\_\_(self):

super(InsuranceCNN, self).\_\_init\_\_()

self.conv1 = nn.Conv2d(in\_channels=3, out\_channels=16, kernel\_size=3, padding=1)

self.pool = nn.MaxPool2d(2, 2)

self.conv2 = nn.Conv2d(16, 32, 3, padding=1)

self.conv3 = nn.Conv2d(32, 64, 3, padding=1)

self.fc1 = nn.Linear(64 \* 16 \* 16, 128)

self.dropout = nn.Dropout(0.3)

self.fc2 = nn.Linear(128, 2) # 2 Klassen: damaged / whole

def forward(self, x):

x = self.pool(F.relu(self.conv1(x))) # 128x128 → 64x64

x = self.pool(F.relu(self.conv2(x))) # 64x64 → 32x32

x = self.pool(F.relu(self.conv3(x))) # 32x32 → 16x16

x = x.view(-1, 64 \* 16 \* 16) # Flatten

x = F.relu(self.fc1(x))

x = self.dropout(x)

x = self.fc2(x) # Kein Softmax – CrossEntropyLoss übernimmt das

return x

# Klassenübersicht

class\_names = train\_dataset.classes

import json

class\_to\_idx = train\_dataset.class\_to\_idx

with open("model/class\_mapping.json", "w") as f:

json.dump(class\_to\_idx, f)

print("📄 Klassen-Index-Mapping gespeichert: model/class\_mapping.json")

# Gerät erkennen: CUDA, MPS (für Apple M1/M2) oder CPU

if torch.backends.mps.is\_available():

device = torch.device("mps")

elif torch.cuda.is\_available():

device = torch.device("cuda")

else:

device = torch.device("cpu")

print("🖥️ Verwende Gerät:", device)

model = InsuranceCNN().to(device)

criterion = nn.CrossEntropyLoss()

optimizer = torch.optim.Adam(model.parameters(), lr=0.001)

num\_epochs = 100

for epoch in range(num\_epochs):

model.train()

running\_loss = 0.0

correct = 0

total = 0

for inputs, labels in train\_loader:

inputs, labels = inputs.to(device), labels.to(device)

optimizer.zero\_grad()

outputs = model(inputs)

loss = criterion(outputs, labels)

loss.backward()

optimizer.step()

running\_loss += loss.item()

\_, predicted = torch.max(outputs.data, 1)

total += labels.size(0)

correct += (predicted == labels).sum().item()

train\_acc = 100 \* correct / total

train\_loss = running\_loss / len(train\_loader)

# Validation

model.eval()

val\_correct = 0

val\_total = 0

with torch.no\_grad():

for inputs, labels in val\_loader:

inputs, labels = inputs.to(device), labels.to(device)

outputs = model(inputs)

\_, predicted = torch.max(outputs.data, 1)

val\_total += labels.size(0)

val\_correct += (predicted == labels).sum().item()

val\_acc = 100 \* val\_correct / val\_total

print(f"📊 Epoch [{epoch+1}/{num\_epochs}] "

f"Train Loss: {train\_loss:.4f} | "

f"Train Acc: {train\_acc:.2f}% | "

f"Val Acc: {val\_acc:.2f}%")

# Modell speichern

torch.save(model.state\_dict(), "model/deep\_model.pth")

print("💾 Modell gespeichert: model/deep\_model.pth")