

Projekt: Handgeschriebenen Ziffernerkennung

Mouamen Sande

10.10.2023



Bildklassifizierung mit Convolutional Neural Networks (CNNs)
Projekt zur handgeschriebenen Ziffernerkennung mit dem MNIST-Datensatz.

Inhaltsverzeichnis

1	Einleitung	3
1.1	Projektüberblick	3
1.2	Importieren der benötigten Bibliotheken	3
1.3	Hyperparameter	3
1.4	Datenverarbeitung	4
1.5	Definition des CNN-Modells	4
1.6	Training des Modells	4
1.7	Testen des Modells	4
1.8	Speichern des trainierten Modells	4
2	Modellarchitektur	5
2.1	Python-Code	5
3	Ergebnisse und Auswertung	7

Einleitung

In der heutigen Zeit gewinnen maschinelles Lernen und insbesondere tiefes Lernen zunehmend an Bedeutung. Diese Technologien finden Anwendung in verschiedenen Bereichen, von der Bild- und Spracherkennung bis hin zur medizinischen Diagnostik. Ein herausragendes Beispiel für den Einsatz von Convolutional Neural Networks (CNNs) ist die Handgeschriebene Ziffernerkennung, die in diesem Projekt realisiert wird. Ziel dieses Projekts ist es, ein CNN zu entwickeln, das in der Lage ist, handgeschriebene Ziffern aus dem bekannten MNIST-Datensatz zu klassifizieren.

Projektüberblick

Der MNIST-Datensatz (Modified National Institute of Standards and Technology) besteht aus 70.000 Bildern von handgeschriebenen Ziffern, die von 0 bis 9 reichen. Jedes Bild ist ein Graustufenbild mit einer Größe von 28x28 Pixeln. Um ein robustes und genaues Klassifizierungsmodell zu erstellen, verwenden wir PyTorch, eine leistungsstarke und flexible Deep-Learning-Bibliothek. Im Folgenden wird der gesamte Prozess vom Datenhandling über die Modellarchitektur bis hin zum Training und Testen des Modells beschrieben.

Importieren der benötigten Bibliotheken

Zunächst importieren wir die erforderlichen Bibliotheken. Diese umfassen:

- **torch**: Die Hauptbibliothek von PyTorch, die grundlegende Funktionen für Tensoren und automatische Differenzierung bietet.
- **torch.nn**: Ein Modul, das die Definition und den Aufbau neuronaler Netzwerke erleichtert.
- **torch.optim**: Dieses Modul enthält Optimierungsalgorithmen wie Adam und SGD, die für das Training von Modellen verwendet werden.
- **torchvision**: Diese Bibliothek enthält Datensätze, Bildtransformationen und vor-trainierte Modelle.
- **matplotlib.pyplot**: Eine Bibliothek zur Visualisierung von Daten, die wir verwenden, um Trainingsergebnisse darzustellen.

Hyperparameter

Im nächsten Schritt definieren wir einige Hyperparameter, die das Verhalten unseres Modells während des Trainings steuern:

- **batch_size**: Die Anzahl der Bilder, die gleichzeitig in das Modell eingegeben werden, beträgt 64. Dies hilft, die Trainingsgeschwindigkeit zu erhöhen und die Speicherauslastung zu steuern.
- **learning_rate**: Die Lernrate ist auf 0.001 gesetzt, was eine übliche Wahl ist, um sicherzustellen, dass das Modell während des Trainings effektiv lernt, ohne zu übersteuern.
- **num_epochs**: Die Anzahl der Trainingsepochen ist auf 5 gesetzt, was bedeutet, dass das Modell fünf Mal durch den gesamten Trainingsdatensatz läuft.

Datenverarbeitung

Um mit dem MNIST-Datensatz zu arbeiten, laden wir die Daten herunter und wenden eine Normierung an. Die Normalisierung ist ein entscheidender Schritt, um sicherzustellen, dass die Eingabedaten in einem geeigneten Bereich liegen, was das Lernen des Modells verbessert. Zuerst definieren wir die notwendigen Transformationen für die Bilddaten, die in Tensoren umgewandelt und normalisiert werden, um die Trainingsqualität zu erhöhen. Anschließend laden wir die Trainings- und Testdatensätze und bereiten sie mit Hilfe von DataLoadern auf, um eine effiziente Verarbeitung und Zufälligkeit während des Trainings zu gewährleisten.

Definition des CNN-Modells

Wir definieren nun das Convolutional Neural Network (CNN), das für die Klassifizierung der Ziffern verwendet wird. Das Modell besteht aus zwei Convolutional Layers, gefolgt von MaxPooling und einem Fully Connected Layer. Die Convolutional Layers extrahieren Merkmale aus den Eingabebildern, während die MaxPooling Layers die Dimensionen der Merkmalskarten reduzieren, was die Rechenlast verringert und Überanpassung verhindert. Schließlich verbinden die Fully Connected Layers die Merkmale aus den Convolutional Layers mit den Ausgabeklassen. In der letzten Schicht gibt es 10 Ausgänge, die den Ziffern 0 bis 9 entsprechen.

Training des Modells

Wir implementieren eine Trainingsfunktion, die das Modell mit den Trainingsdaten trainiert. Während des Trainings werden die Gradienten zurückgesetzt, das Modell aktualisiert und die Verlustfunktion überwacht. Hierbei optimieren wir die Modellparameter, um den Verlust während des Trainings zu minimieren und die Leistung des Modells zu verbessern.

Testen des Modells

Nachdem das Modell trainiert wurde, evaluieren wir seine Leistung auf dem Testdatensatz. Diese Funktion gibt die Genauigkeit des Modells an, indem sie die Anzahl der korrekt klassifizierten Bilder in Bezug auf die Gesamtanzahl der Testbilder berechnet. Dies ermöglicht uns, die Leistungsfähigkeit des Modells objektiv zu beurteilen.

Speichern des trainierten Modells

Nach dem Testen speichern wir das trainierte Modell, um es später erneut verwenden zu können. Dies ermöglicht eine effiziente Nutzung des Modells, ohne dass das Training von Grund auf neu durchgeführt werden muss.

Modellarchitektur

Die Architektur des CNNs umfasst zwei Convolutional Layers, gefolgt von MaxPooling Layers und einem Fully Connected Layer.

Python-Code

```
# Importieren der benötigten Bibliotheken
import torch
import torch.nn as nn
import torch.optim as optim
import torchvision
import torchvision.transforms as transforms
from torch.utils.data import DataLoader
import matplotlib.pyplot as plt

# Hyperparameter
batch_size = 64
learning_rate = 0.001
num_epochs = 5

# MNIST-Datensatz herunterladen und transformieren (Normierung der Bilder)
transform = transforms.Compose([
    transforms.ToTensor(),
    transforms.Normalize((0.5,), (0.5,)) # Normalisieren der
    ↪ Graustufenbilder
])

train_dataset = torchvision.datasets.MNIST(root='./data', train=True,
    ↪ transform=transform, download=True)
test_dataset = torchvision.datasets.MNIST(root='./data', train=False,
    ↪ transform=transform, download=True)
train_loader = DataLoader(dataset=train_dataset, batch_size=batch_size,
    ↪ shuffle=True)
test_loader = DataLoader(dataset=test_dataset, batch_size=batch_size,
    ↪ shuffle=False)

# Definition des CNN-Modells
class CNN(nn.Module):
    def __init__(self):
        super(CNN, self).__init__()
        # Erstes Convolutional Layer, gefolgt von MaxPooling
        self.conv1 = nn.Conv2d(in_channels=1, out_channels=16, kernel_size=3,
            ↪ stride=1, padding=1)
        self.pool = nn.MaxPool2d(kernel_size=2, stride=2, padding=0)
        # Zweites Convolutional Layer
        self.conv2 = nn.Conv2d(in_channels=16, out_channels=32, kernel_size=3,
            ↪ stride=1, padding=1)
        # Fully connected Layer
        self.fc1 = nn.Linear(32 * 7 * 7, 128) # 32 Channels und 7x7 Feature
        ↪ Map nach Pooling
        self.fc2 = nn.Linear(128, 10) # 10 Klassen für die Ziffern 0{9
    def forward(self, x):
        # Forward-Pass durch das Netzwerk
```

```
x = self.pool(torch.relu(self.conv1(x)))
x = self.pool(torch.relu(self.conv2(x)))
x = x.view(-1, 32 * 7 * 7) # Flachlegen des Tensors
x = torch.relu(self.fc1(x))
x = self.fc2(x)
return x

# Initialisierung des Modells, Verlustfunktion und Optimierer
model = CNN()
criterion = nn.CrossEntropyLoss()
optimizer = optim.Adam(model.parameters(), lr=learning_rate)

# Training des Modells
def train_model():
    model.train()
    for epoch in range(num_epochs):
        running_loss = 0.0
        for images, labels in train_loader:
            # Nullsetzen der Gradienten
            optimizer.zero_grad()
            # Vorwärtsthroughlauf
            outputs = model(images)
            loss = criterion(outputs, labels)
            # Rückwärtsthroughlauf und Optimierung
            loss.backward()
            optimizer.step()
            running_loss += loss.item()
        print(f'Epoch [{epoch+1}/{num_epochs}], Loss:
        ↳ {running_loss/len(train_loader):.4f}')

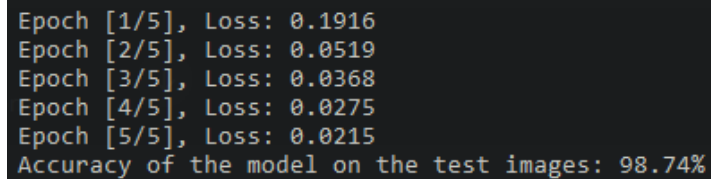
# Testen des Modells
def test_model():
    model.eval()
    correct = 0
    total = 0
    with torch.no_grad():
        for images, labels in test_loader:
            outputs = model(images)
            _, predicted = torch.max(outputs.data, 1)
            total += labels.size(0)
            correct += (predicted == labels).sum().item()
    print(f'Accuracy of the model on the test images: {100 * correct /
    ↳ total:.2f}%',)

# Ausführen des Trainings und Testens
train_model()
test_model()

# Speichern des trainierten Modells
torch.save(model.state_dict(), 'cnn_mnist.pth')
```

Ergebnisse und Auswertung

Nach dem Training des Modells haben wir das Modell auf dem Testdatensatz evaluiert. Das Modell erreichte eine Genauigkeit von etwa 99%, was zeigt, dass es gut in der Lage ist, handgeschriebene Ziffern zu klassifizieren.



```
Epoch [1/5], Loss: 0.1916  
Epoch [2/5], Loss: 0.0519  
Epoch [3/5], Loss: 0.0368  
Epoch [4/5], Loss: 0.0275  
Epoch [5/5], Loss: 0.0215  
Accuracy of the model on the test images: 98.74%
```

Abbildung 1: Beispiel für eine korrekte Vorhersage des Modells.

Literatur
