CNN für Handwritten Digits Recognition

Ange Nguetsop

February 2, 2024

1 Introduction

Handgeschriebene Ziffernerkennung ist eine wichtige Aufgabe in der Bildverarbeitung und im maschinellen Lernen. Sie findet Anwendung in verschiedenen Bereichen wie Postleitzahlenlesung, Bankenwesen, und digitale Unterschriftenverifizierung. Ein Convolutional Neural Network (CNN) ist eine leistungsfähige Methode zur automatisierten Erkennung von handgeschriebenen Zahlen.

In diesem System verwenden wir ein CNN, um handgeschriebene Zahlen zu erkennen. Ein CNN ist eine spezielle Art von neuronalem Netzwerk, das sich besonders gut für die Verarbeitung von Bildern eignet. Es besteht aus mehreren Schichten von Neuronen, die so angeordnet sind, dass sie Informationen aus den Bildern aufnehmen, Merkmale extrahieren und schließlich die erkannten Zahlen klassifizieren können.

2 CNN für Handwritten Digits Recognition

2.1 First Step: Import Dataset

Die erste Phase umfasst die Importierung der notwendigen PyTorch-Bibliotheken für unser Modell. Dazu gehören unter anderem:

- **torchvision**: Diese Bibliothek enthält Datensätze, Modellarchitekturen und Bildtransformationen, die häufig für Computer Vision-Probleme verwendet werden.
- **torchvision.datasets**: Hier finden Sie viele Beispiel-Datensätze für verschiedene Computer Vision-Probleme wie Bildklassifikation, Objekterkennung, Bildbeschriftung, Videoklassifikation und mehr. Es enthält auch eine Reihe von Basisklassen zur Erstellung eigener Datensätze.
- **torchvision.models**: Dieses Modul enthält gut performende und häufig verwendete Modellarchitekturen für Computer Vision, die in PyTorch implementiert sind. Sie können diese Modelle für Ihre eigenen Probleme verwenden.
- **torchvision.transforms**: Oft müssen Bilder transformiert werden, bevor sie mit einem Modell verwendet werden können. Hier finden Sie gängige Bildtransformationen wie Umwandlungen in Zahlen, Verarbeitung und Augmentierung.
- **torch.utils.data.Dataset**: Diese Basisklasse dient als Grundlage für die Erstellung eigener Datensätze in PyTorch.
- **torch.utils.data.DataLoader**: Dieses Modul erstellt eine Python-Iteration über einen Datensatz, der mit `torch.utils.data.Dataset` erstellt wurde.

Diese Bibliotheken sind wesentlich, um Computer-Vision-Modelle in PyTorch zu entwickeln und umfassen Datensätze, Modelle, Transformationen und Tools zur effizienten Verwaltung von Daten. Mit ihrer Hilfe können Sie verschiedene Computer-Vision-Probleme lösen, angefangen bei der Datenbeschaffung bis zur Modellerstellung und -auswertung.

```
# Import PyTorch
import torch
from torch import nn

# Import torchvision
import torchvision
from torchvision import datasets
from torchvision.transforms import ToTensor

# Import matplotlib for visualization
import matplotlib.pyplot as plt
```

Listing 1: Import Libraries

Die nächste Phase umfasst das Herunterladen unseres Datensatzes. Der Datensatz wird in zwei Teile aufgeteilt: 80 % für die Trainingsphase und 20 % für die Testphase.

```
train_data = datasets.MNIST(
    root = 'data',
    train = True,
    download = True,
    transform = ToTensor()
)

test_data = datasets.MNIST(
    root = 'data',
    train = False,
    download = True,
    transform = ToTensor()
)
```

Listing 2: Training and Test Dataset

2.2 Umwandlung Daten in Batches

Jetzt haben wir einen einsatzbereiten Datensatz. Der nächste Schritt besteht darin, ihn mit einem `torch.utils.data.DataLoader` oder kurz `DataLoader` vorzubereiten. Der `DataLoader` tut, was Sie vielleicht denken. Er hilft dabei, Daten in ein Modell zu laden. Sowohl für das Training als auch für die Inferenz. Er verwandelt einen großen Datensatz in ein Python-Iterable aus kleineren Teilen. Diese kleineren Teile werden als Batches oder Mini-Batches bezeichnet und können mit dem Parameter `batch_size` festgelegt werden.

```
from torch.utils.data import DataLoader

batchSize = 100

train_dataloader = DataLoader(
    dataset = train_data,
    batch_size = batchSize,
    shuffle = True,
    num_workers=1
)

test_dataloader = DataLoader(
    dataset = test_data,
    batch_size = batchSize,
    shuffle = False,
    num_workers=1

)
```

Listing 3: Split Data into Batches

2.3 Initialisierung des CNNs

Jetzt ist es an der Zeit, unser Modell zu initialisieren. Die Initialisierung eines solchen Modells ist ziemlich einfach. Die Daten durchlaufen zunächst die erste Schicht, die aus einem Convolutional Layer, einem Pooling Layer und einer ReLU-Funktion besteht, um Nichtlinearität in das Modell einzuführen.

Dann durchlaufen die Daten die zweite Schicht, indem sie nacheinander den zweiten Convolutional Layer, einen Dropout Layer zur Regularisierung, um Overfitting in neuronalen Netzwerken zu reduzieren, durchlaufen, bevor sie erneut einen Pooling Layer und eine ReLU-Funktion durchlaufen. Anschließend durchlaufen die Daten zwei Fully Connected Layer, bevor sie schließlich Vorhersagen treffen.

```
import torch.nn.functional as F
import torch.optim as optim

class MNISTModel(nn.Module):

  def __init__(self):
    super(MNISTModel, self).__init__()

    self.conv1 = nn.Conv2d(1,10,kernel_size=5)
    self.conv2 = nn.Conv2d(10,20,kernel_size=5)
    self.conv2_drop = nn.Dropout2d()
    self.fc1 = nn.Linear(320,50)
    self.fc2 = nn.Linear(50,10)


  def forward(self,x):
    x = F.relu(F.max_pool2d(self.conv1(x),2))
    x = F.relu(F.max_pool2d(self.conv2_drop(self.conv2(x)),2))
    x = x.view(-1,320)
    x = F.relu(self.fc1(x))
    x = F.dropout(x, training = self.training)
    x = self.fc2(x)

    return F.softmax(x)
```

Listing 4: Split Data into Batches

2.4 Loss function and Optimizer

Wir werden die gleichen Funktionen wie zuvor verwenden: `nn.CrossEntropyLoss()` als Verlustfunktion (da wir mit mehrklassigen Klassifikationsdaten arbeiten) und `torch.optim.SGD()` als Optimierer, um `model_2.parameters()` mit einer Lernrate von 0,001 zu optimieren.

```
import requests
from pathlib import Path

request = requests.get("https://raw.githubusercontent.com/mrdbourke/pytorch-deep-
    learning/main/helper_functions.py")
with open("helper_functions.py", "wb") as f:
    f.write(request.content)


from helper_functions import accuracy_fn
from tqdm.auto import tqdm

model = MNISTModel().to(device)

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

loss_fn = nn.CrossEntropyLoss()
```

Listing 5: Split Data into Batches

2.5 Functions for the Training und testing

Nun definieren wir Funktionen für die Training- und Testschritte.

```
device = "cuda" if torch.cuda.is_available() else "cpu"
def train_step(model: torch.nn.Module,
               data_loader: torch.utils.data.DataLoader,
               loss_fn: torch.nn.Module,
               optimizer: torch.optim.Optimizer,
```

```
              accuracy_fn,
              device: torch.device = device):
    train_loss, train_acc = 0, 0
    model.to(device)
    for batch, (X, y) in enumerate(data_loader):
        # Send data to GPU
        X, y = X.to(device), y.to(device)

        # 1. Forward pass
        y_pred = model(X)

        # 2. Calculate loss
        loss = loss_fn(y_pred, y)
        train_loss += loss
        train_acc += accuracy_fn(y_true=y,
                                 y_pred=y_pred.argmax(dim=1)) # Go from logits -> pred
     labels

        # 3. Optimizer zero grad
        optimizer.zero_grad()

        # 4. Loss backward
        loss.backward()

        # 5. Optimizer step
        optimizer.step()

    # Calculate loss and accuracy per epoch and print out what's happening
    train_loss /= len(data_loader)
    train_acc /= len(data_loader)
    print(f"Train loss: {train_loss:.5f} | Train accuracy: {train_acc:.2f}%")

def test_step(data_loader: torch.utils.data.DataLoader,
              model: torch.nn.Module,
              loss_fn: torch.nn.Module,
              accuracy_fn,
              device: torch.device = device):
    test_loss, test_acc = 0, 0
    model.to(device)
    model.eval() # put model in eval mode

    # Turn on inference context manager
    with torch.inference_mode():
        for X, y in data_loader:
            # Send data to GPU
            X, y = X.to(device), y.to(device)

            # 1. Forward pass
            test_pred = model(X)

            # 2. Calculate loss and accuracy
            test_loss += loss_fn(test_pred, y)
            test_acc += accuracy_fn(y_true=y,
                y_pred=test_pred.argmax(dim=1) # Go from logits -> pred labels
            )

        # Adjust metrics and print out
        test_loss /= len(data_loader)
        test_acc /= len(data_loader)
        print(f"Test loss: {test_loss:.5f} | Test accuracy: {test_acc:.2f}%\n")
```

Listing 6: Function for Training and Testing

Jetzt können wir mit dem Training beginnen.

```
# Train and test model
epochs = 50
for epoch in tqdm(range(epochs)):
    print(f"Epoch: {epoch}\n---------")
    train_step(data_loader=train_dataloader,
        model=model,
        loss_fn=loss_fn,
```

```
        optimizer=optimizer,
        accuracy_fn=accuracy_fn,
        device=device
    )
    test_step(data_loader=test_dataloader,
        model=model,
        loss_fn=loss_fn,
        accuracy_fn=accuracy_fn,
        device=device
    )
```

Listing 7: Train and Test model

2.6 Predictions with the model

```
# Import tqdm for progress bar
from tqdm.auto import tqdm

# 1. Make predictions with trained model
y_preds = []
model.eval()
with torch.inference_mode():
  for X, y in tqdm(test_dataloader, desc="Making predictions"):
    # Send data and targets to target device
    X, y = X.to(device), y.to(device)
    # Do the forward pass
    y_logit = model(X)
    # Turn predictions from logits -> prediction probabilities -> predictions labels
    y_pred = torch.softmax(y_logit, dim=1).argmax(dim=1) # note: perform softmax on
    the "logits" dimension, not "batch" dimension (in this case we have a batch size of
     32, so can perform on dim=1)
    # Put predictions on CPU for evaluation
    y_preds.append(y_pred.cpu())
# Concatenate list of predictions into a tensor
y_pred_tensor1 = torch.cat(y_preds)
```

Listing 8: Predictions with the model

2.7 Confusion Matrix

Eine Confusion Matrix (Verwirrungsmatrix) ist ein häufig verwendetes Werkzeug zur Evaluierung von Klassifikationsalgorithmen, insbesondere in maschinellem Lernen und Data Science. Sie bietet eine Möglichkeit, die Leistung eines Klassifikators zu visualisieren und zu quantifizieren.

Eine Confusion Matrix ist eine quadratische Tabelle, die die tatsächlichen Klassen (wahre Positiven, wahre Negativen, falsche Positiven und falsche Negativen) gegenüber den vom Klassifikator vorhergesagten Klassen darstellt. Sie enthält normalerweise vier wichtige Werte:

- **True Positives (TP)**: Die Anzahl der Instanzen, bei denen der Klassifikator korrekt die positive Klasse vorhergesagt hat.
- **True Negatives (TN)**: Die Anzahl der Instanzen, bei denen der Klassifikator korrekt die negative Klasse vorhergesagt hat.
- **False Positives (FP)**: Die Anzahl der Instanzen, bei denen der Klassifikator fälschlicherweise die positive Klasse vorhergesagt hat, obwohl es sich um die negative Klasse handelt. Dies wird auch als Typ-I-Fehler bezeichnet.
- **False Negatives (FN)**: Die Anzahl der Instanzen, bei denen der Klassifikator fälschlicherweise die negative Klasse vorhergesagt hat, obwohl es sich um die positive Klasse handelt. Dies wird auch als Typ-II-Fehler bezeichnet.

```
# See if torchmetrics exists, if not, install it
try:
    import torchmetrics, mlxtend
```

```
    print(f"mlxtend version: {mlxtend.__version__}")
    assert int(mlxtend.__version__.split(".")[1]) >= 19, "mlxtend verison should be
    0.19.0 or higher"
except:
    !pip install -q torchmetrics -U mlxtend # <- Note: If you're using Google Colab,
    this may require restarting the runtime
    import torchmetrics, mlxtend
    print(f"mlxtend version: {mlxtend.__version__}")

# Import mlxtend upgraded version
import mlxtend
print(mlxtend.__version__)
assert int(mlxtend.__version__.split(".")[1]) >= 19
# should be version 0.19.0 or higher

from torchmetrics import ConfusionMatrix
from mlxtend.plotting import plot_confusion_matrix

# 2. Setup confusion matrix instance and compare predictions to targets
confmat = ConfusionMatrix(num_classes=len(class_names), task='multiclass')
confmat_tensor = confmat(preds=y_pred_tensor1,
                         target=test_data.targets)

# 3. Plot the confusion matrix
fig, ax = plot_confusion_matrix(
    conf_mat=confmat_tensor.numpy(), # matplotlib likes working with NumPy
    class_names=class_names, # turn the row and column labels into class names
    figsize=(10, 7)
);
```

Listing 9: Predictions with the model

Hier ist unser Confusion Matrix:

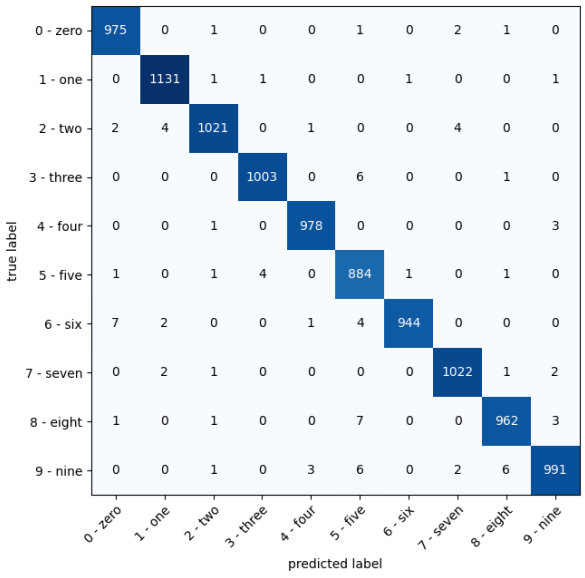

Figure 1: Testergebnisse

Wir können anhand unserer Konfusionsmatrix sehen, dass unser Modell nicht in der Lage ist, eine Zahl mit einer Genauigkeit von 100 % vorherzusagen. Sieben kann zum Beispiel als 1 oder als 9 vorhergesagt werden. Aber im Allgemeinen haben wir ein ziemlich gutes Modell.

2.8 Test on real Data

Und nun ist es an der Zeit, unser Modell an echten Daten zu testen. Wir können sehen, dass unser Modell wirklich ein Problem mit der Vorhersage von eins und sieben hat.

Figure 2: Testergebnisse