Reflexion

Abstrakt

Dieses Projekt handelt von einem komplexen Computerprogramm, welches, durch vieles lernen, ein Muster in Zeichnungen erkennen kann. Durch diese Muster kann das Programm, nebst den fünfzig Millionen Zeichnungen, auf welche es trainiert wurde, auch ganz neue Zeichnungen erkennen und dann eine Aussage macht, was auf diesem Bild zu sehen ist. Dieses Computerprogramm ist in meinem Fall ein CNN (Convolutional Neural Network) welches mit Pytorch geschrieben wurde.

Prozess

Ideen finden

Dieses Projekt wurde im Oktober 2022 gestartet und dauerte bis zum Ende Januar 2023. Während dieser Zeit habe ich vieles gelernt und mein Projekt wurde deutlich grösser als zuerst gedacht. Ich wollte zu Beginn einfach einen «einfaches» Neuronales Netz schreiben, welches eine gewisse Anzahl an Kunstwerken zuordnen kann. Ich brauchte für das Finden eines geeigneten Datenset ungefähr zehn Stunden. Der gefundene Datensatz hatte einen grossen Einfluss auf den weiteren Verlauf dieses Projektes. Am Ende meiner Suchzeit bin ich auf das Datenset von QuickDraw gestossen (1, 2). Dies ist eines der grössten Datensets für schnelle Zeichnungen, auch Doodles genannt. Es stammt von einem online Multiplayer-Spiel. Es hat einen Umfang von ungefähr fünfzig Millionen Zeichnungen, welche immer von den SpielerInnen überarbeitet und korrigiert werden.

Tutorial zu QuickDraw

Als ich dieses Datenset gefunden hatte, war ich erstaunt, da es auch noch ein Tutorial für ein Neuronales Netz gab (3). Ich wollte diesen Code nutzen, damit mein Programm trainieren und dann eine Aplikation darum schreiben, welche Interaktion damit vereinfachen würde. Dies versuchte ich auch für ungefähr fünf Stunden, wobei ich zugeben muss, dass ich in Verzweiflung später auch noch ein paar Mal darauf zurück kam. Das Problem mit dem Programm war, dass es im Jahre 2017 geschrieben wurde, und die genutzten Libraries (hier Tensorflow, 4) seitdem überarbeitet worden sind und somit nicht mehr funktionieren. Ausserdem habe ich nie gelernt mit Tensorflow umzugehen und konnte die Fehler somit auch nicht korrigieren. Wegen diesen Problemen musste ich diesen Weg aufgeben.

Beginn meines Eigenen Programmes (NDJSON)

Als zweites wollte ich mein eigenes Programm schreiben, dieses Mal jedoch mit PyTorch (5). Ich begann dies Anfang Dezember und machte zu Beginn viel fortschritt. Ich hatte die allgemeine Struktur des Models, welche sich in dieser kurzen Zeit von einem «normalen» Natural Neural Network (NNN) zu einem Recurrent Neural Network (RNN) entwickelt hatte. Diese Modelle will ich hier nicht weiter erläutern, jedoch verlinke ich Sie gerne auf eine erklärende Website (6). Diese Netzwerke wollte ich mit den Daten im Format Ndjson (Newline delimeted JavaScript Object Notation) füttern, was jedoch auf allen Seiten zu Problemen führte. Dies hatte mehrere Gründe. Ein grosser Grund hierfür war, dass Ndjson ein neueres Datenformat ist, somit wenig Dokumentation hat und auch nicht intuitiv zum Verstehen beziehungsweise anwenden ist. Ich hatte dort für sehr lange Zeit immer den gleichen

Fehler, welcher sich weder durch mehrmaliges komplett neuschreiben des Programmes noch durch Recherchen oder auch Fragen auf online Foren lösen lies. Insgesamt habe ich in diesem Abschnitt des Programmes ungefähr zwanzig Stunden zugebracht, ohne wirklich Resultate zu sehen. Ich habe in dieser Zeit jedoch sehr viel über den Aufbau des RNN und über den Umgang mit Daten gelernt, welches mir bei meinem nächsten Abschnitt sicher geholfen hatte.

Mein eigenes Programm (NPY)

In diesem Abschnitt habe ich mich entschieden mein Programm mit den NDJSON-Daten zurückzulassen und ein neues mit NPY-Daten (Datenformat von der Library NumPy, 7) zu versuchen. Dieser Datentyp ist deutlich unübersichtlicher, da er in einem Texteditor nicht gelesen werden kann, jedoch hat es deutlich mehr Dokumentation dazu und ich hatte schon zuvor damit gearbeitet. Hier versuchte ich zuerst mein Programm von zuvor zu verwenden, in dem ich es ein bisschen umschreiben würde, jedoch habe ich nach einiger Zeit gemerkt, dass dies keine Option war, da die Daten komplett anders organisiert waren. Die Daten in den NDJSON-Daten hatten zu Beginn zuerst die Zahl welche diesem Objekt zugeordnet wurde, dann das Label, also die Klasse oder der Name des Objektes in der Zeichnung, darauf folgend den Namen des Landes, von wo es gezeichnet wurde, darunter die Zeitangabe zu der Zeichnung, also wann sie kreiert worden war, danach eine Wahr Falsch Aussage, ob das Objekt erkannt wurde und dann eine Liste von Zahlen in JSON-Format, welche die x, y und t (Zeit seit dem ersten ansetzen des Stiftes) Achsen repräsentierten. Dies kann auch auf der Website gesehen werden (2). Die Daten der NumPy-Daten waren einfach eine Pixelangabe für ein 28 Pixel hohes und 28 Pixel breites Bild.

Entwicklung des trainierten Models

In diesem Abschnitt meiner Reise habe ich sicherlich am meisten Zeit verbracht. Ich habe den Code neu geschrieben, die neuen Probleme bestritten und mir eine komplett neue Fähigkeit im Bezug auf das Programmieren anschaffen dürfen. Diesen Teil der Arbeit geschah fast ausschliesslich in den Winterferien, von Neujahr bis zum Anfang der Schule. In dieser Woche hatte ich mehr als genug Zeit und Lust an diesem Projekt zu arbeiten. Ich denke stark, dass das ständige Programmiren und Fehlerbeheben über mehrere Tage hinweg eine der besten Trainingsmöglichkeiten waren. Für einige Tage habe ich nichts anderes gemacht als an dem Programm zu programmieren. Dies führte dazu, dass ich bei jeder freien Gelegenheit daran dachte und auch davon träumte. Diese Ideen waren meistens die besten, da sie mit einem anderen Blickwinkel auf das Projekt entstanden sind.

Während diesen fünf Tagen habe ich es hingekriegt, um die dreissig Stunden an meinem Projekt zu arbeiten und habe somit auch einen funktionierenden Code vorzuweisen gehabt. Diesen Code habe ich am 07.01.2023 fertiggestellt und dann vom 08.01.2023 bis am 15.01.2023 trainieren lassen. Alles in allem hat das Training ungefähr 150 Stunden in Anspruch genommen. Dies führte dazu, dass ich für eine Woche mit einem laufen Computer in meinem Zimmer geschlafen habe. Während dieser Trainingsphase gab es einige Schwierigkeiten mit automatischen Windowsupdates und einem Absturz meines PC, jedoch habe ich durch diese Unannehmlichkeiten nur ungefähr zehn Stunden an Training verloren, da ich das Programm zwischendurch immer wieder speichere.

Die finale Idee hinter dem Programm

Während der obigen Phase habe ich versucht meine schon Gespeicherten Netzwerke zu nutzen und ein Programm zu schreiben, welches eigene Zeichnungen klassifizieren kann. Hierfür habe ich vor allem Pygame verwendet (10). Nach einigem Einlesen in die Pygame Bibliothek habe ich es auch hin gekriegt mehrere Fenster zu öffnen und eine Eingabe in diesen zu machen. Ich konnte in diesem Programm eine Gruppe auswählen, dann ein Bild dazu zeichnen, dieses dann speichern und dann durch mein Neuronales Netz zu führen. Das funktionierte auch nach einigen Anläufen, wobei es doch

einen deutlichen Fehler gab. Das Problem war, dass ich auf einer riesigen Fläche gezeichnet hatte, diese Zeichnung dann gespeichert und zu einer 28 mal 28 NumPy Datei verwandelte. Durch dieses deutliche und auch unvorsichtige verkleinern des Bildes werden die Ränder verschwommen und sehen nicht so aus wie die Daten, auf welches das Programm trainiert war. Dies führt nun dazu, dass dieses die Bilder nicht richtig klassifizieren kann, da es die Bilder nicht erkennen kann. Ich hätte dieses Programm gerne weitergeführt, jedoch habe ich leider keine Zeit dazu.

Die Zukunft des Programmes

Dieses Programm hätte zu einem Spiel weiterentwickelt werden können, in welchem zwei Spieler gegeneinander zeichnen. Das Programm wurde auf eine riesige Datenmenge trainiert, was dazu führte, dass es sich die ähnlichsten Bilder fast auswendig gemerkt hatte und diese auch fast immer klassifizieren kann. Dies könnte man für ein Spiel nutzen, welches zum Beispiel «Draw Generic» heissen könnte. Die Person gewinnt, welche mehr erkannte Bilder durch das CNN hat und somit durchschnittlicher gezeichnet hatte.

Eine weitere Anwendungsmöglichkeit hätte es beim digitalen Montagsmaler. Dort könnte man, falls man gerne mehr Mitspieler haben möchte, das Model als weitere Person nehmen. Das Programm würde während des Zeichnens schon die ganze Zeit Vorhersagen machen und wäre somit genau gleich wie die anderen Mitspieler.

Dieses Programm könnte auch dazu verwendet werden, um weitere Daten einzuordnen. Dies wäre sehr hilfreich, da man dies bei zu hohen Datenmengen nicht mehr von Hand machen kann.

Produkt

Erreicht

Für mich ist das Grösste, was ich während dieses Projektes erreicht habe, die Menge an Wissen, welche ich gewonnen habe. Ich kann nun Programme schreiben, welche ich zuvor nicht einmal verstanden hatte. Durch dieses Projekt habe ich auch erreicht, dass ich nun meine Passion für das Programmieren definitiv gefunden habe.

Durch mein Programm habe ich es erreicht über fünfzig Millionen Bilder mit einer durchschnittlichen Genauigkeit von 75% zu klassifizieren. Dies sind 38 Millionen richtig zugeordnete Bilder.

Dies ist nicht allzu nützlich, jedoch sind durch dieses richtigen Klassifizierungen Dateien entstanden, welche die Gewichtung des CNN gespeichert haben. Diese Dateien können von allen verwendet werden, um entweder ihr weiterführendes Programm mit dieser Klassifikatoren zu schreiben oder um das Model weiter zu trainieren. Ich habe hier den grössten oder sogar den ganzen Teil der nötigen Rechenleistung übernommen, was es somit auch schwächeren Computern erlauben wird ein ähnliches Projekt mit den schon vortrainierten Modellen zu schreiben

Ich habe es hingekriegt eine halbwegs UI-Basierte Applikation zu schreiben welche diese Modelle lädt und versucht durch sie eine Klassifikation zu machen.

Verbesserungsmöglich

Ich habe mir in diesem Projekt wie immer viel zu viel vorgenommen. Ich habe mich wie immer komplett überschätzt und die Schwierigkeit unterschätzt. Ich dachte zuerst, dass ich einfach den Code des Tutorials nehmen könnte und dann darauf ein UI-Programm zu schreiben. Dies war bei weitem

nicht der Fall, jedoch lies ich mich auf die Gelegenheit und Challenge ein und erschuf mein bei weitem bestes Programm. Diese Überschätzung führte auch dazu, dass dies eine Maturitätsarbeit hätte sein können, da ich über fünfzig Stunden daran gearbeitet habe und auch sicherlich genug zum Schreiben hätte. Alle diese Fehler hatten jedoch nur gutes mit sich gezogen.

Wie immer gibt es jedoch auch noch negative Seiten, welche hier ohne Frage das nicht vollständige Abschliessen des finalen Programmes war. Hier hatte ich zu wenig Zeit, um dies zu erreichen, auch wenn ich dies in Zukunft sicher noch machen werde.

Alles in allem sollte sich meine Planung verbessern und ich muss viel mehr Zeit für Unvorhergesehenes einplanen. Ich hoffe, dass ich von meinen Fehlern hier für die Maturaarbeit etwas gelernt habe.

Link Verzeichnis

1. QuickDraw Website: https://quickdraw.withgoogle.com/data
2. QuickDraw Daten: https://github.com/googlecreativelab/quickdraw-dataset
3. QuickDraw Tutorial: https://github.com/tensorflow/docs/blob/master/site/en/r1/tutorials/sequences/recurrent_quickdraw.md
4. Tensorflow: https://www.tensorflow.org
5. PyTorch: https://pytorch.org
6. Erklärung Neuronaler Netzwerke, im spezifischen RNN: https://www.simplilearn.com/tutorials/deep-learning-tutorial/rnn#:~:text=RNN%20works%20on%20the%20principle,the%20output%20of%20the%20layer.&text=The%20nodes%20in%20different%20layers,layer%20of%20recurrent%20neural%20networks.
7. NumPy: https://numpy.org/devdocs/index.html
8. Pygame: https://www.pygame.org/news

Anhang

Dieser Anhang soll dem genaueren Verständnis meiner Arbeit dienen. Ich empfehle ihnen sich mit diesem Teil hier auseinander zu setzten, da das Produkt mit dieser Hilfe besser verstanden werden kann.

Erklärung des CNN

Aus dieser Realisation folgte, dass ich das Programm noch einmal neu schreiben musste, und dieses Mal nicht mit einem RNN oder NNN, sondern einem Convolutional Neural Network (CNN), da dieses Model spezifisch für Bilder gedacht ist. Da dies ein wichtiger Teil meiner Arbeit ist, werde ich versuchen die Funktion dieses Models zu erklären. Ein CNN besteht aus mehreren Schichten, durch welche die Daten, in unserem Fall in Form einer Matrix durchgegeben werden. Diese Schichten haben verschiedene Namen, eine wichtige Schicht ist die Convolutional-Schicht, von dort kommt auch der Name des Netzwerkes. Mein Code für diese Schicht sieht so aus:

```
self.conv1 = nn.Conv2d(in_channels=1, out_channels=16, kernel_size=3, padding=1)
```

self.conv1 ist einfach der Name dieser Variabel oder Funktion. Dann wird die schon vorprogrammierte Funktion der Library nn.Conv2d verwendet, um über die Daten zu iterieren. Diese Funktion (Erklärung: https://pytorch.org/docs/stable/generated/torch.nn.Conv2d.html) braucht jedoch noch einige andere Eingaben, hier die in_channels, also die Anzahl an Kanälen, welche von dem Bild erwartet und auch eingegeben werden. Die Anzahl an Kanälen, welche ausgegeben werden, wird hier mit 16 definiert und dies wird weiter durch das Netzwerk gegeben. Die Grösse des Kernels ist zwischen 2 oder 3 zu definieren und ist hier auf 3 gesetzt, damit das Training ein bisschen schneller geht. Diese Grösse ist für die Anzahl an Pixeln gedacht, welche auf einmal von dem CNN verarbeitet werden müssen. Da ich dies hier als 3 definiert habe, sind es drei mal drei Pixel, also eine Fläche von neun Pixeln, welche von links nach rechts, immer mit einem Pixel als Schritt, über das Bild wandern und dieses so umwandeln. Das Padding dient dem Verhalten des Programmes, wenn der Prozess am Rand des Bildes ankommt und ist meistens auf null gestellt. Zusammengefasst ist diese Funktion dazu da das Bild zu analysieren, spezielle Merkmale zu erkennen und diese dann als sogenannte Feature-Maps auszugeben.

Dies ist eine der vielen Funktionen und Variablen, welche in dem CNN definiert werden. Ich habe insgesamt drei Convolutional Schichten, bei welchen die in_channels immer die Ausgaben der Vorherigen sind. Auf diese Funktion folgen noch viele weitere, jedoch werde ich nur noch einige weitere Erklären, da sonst der Umfang gesprengt werden würde. Die Funktion, welche ich als nächstes erläutern werde, ist die Polling-Schicht. Diese Schicht wird meistens zwischen aufeinanderfolgenden Convolutional-Schichten eingebaut und nimmt die Feature-Maps dieser als Eingabe. Mein Code hierfür sieht so aus:

```
self.pool = nn.MaxPool2d(kernel_size=2, stride=2)
```

Die Inputs für diese Funktion sind kernel_size und stride. Diese Eingaben sagen dem CNN zuerst einmal, wie gross die gelesene Fläche des Feature-Maps ist, dies ist hier zwei mal zwei, und wie weit nach rechts es sich bewegen sollte. Dies hätte hier nicht definiert werden müssen, da der normal-Wert für stride gleich der kernel_size ist. Hier sieht man auch, dass ich mich für maxpooling entschieden habe und nicht für das auch weitverbreitete average-pooling. Der Unterschied ist, dass maxpooling den höchsten pixelwert nimmt und average-pooling den Durchschnitt. Da meine Daten hier schwarz-weis sind und ziemlich simple schwarze Kanten haben, habe ich mich für maxpooling entschieden, da das Bild ansonsten verschwommene kannten haben würde. Diese Funktion vermindert somit die Grösse des Bildes, obgleich es doch die wichtigen Merkmale behält.

Zwischen der Pooling-Schicht und der Convolutional-Schicht habe ich noch die ReLu-Funktion eingesetzt. Diese sieht so aus:

self.relu = nn.ReLU()

Wie man hier sehen kann, habe ich keine Inputs gegeben. Diese Funktion hilft dem Neuronalen Netzwerk schneller zu lernen, da es alle negativen Werte gleich Null setzt.

Am Ende des CNN werden noch die Tensoren, das sind eine bestimmte Anzahl an Vektoren in einem Vektor, umgeformt und dann durch die vollverbundenen Schichten gegeben.

self.fc2 = nn.Linear(in_features=128, out_features=num_classes)

Diese Schichten nehmen alle Ausgaben der vorherigen Schichten als Eingaben und macht mit jedem eingabepunkt des Vektors eine Lineare Transformation mit einer Gewichtsmatirx. Hier wird auch die Anzahl der Neuronen so geändert, dass die Neuronen in der letzten Schicht der Anzahl an Klassen entsprechen. Durch diese Schichten werden jegliche Verbindungen und Annahmen zwischen den Klassen gestrichen, wodurch es man diesen Vektor ohne weiteres durch die letzte Funktion des CNN geben kann.

Diese letzte Schicht ist in meinem Fall die Softmax-Schicht. Diese bewirkt, dass zu jedem Vektor, welcher ein Bild repräsentierte, ein Vektor gebildet wird. In diesem werden die Wahrscheinlichkeiten aufgelistet, mit welcher sie zu jeder Klasse gehören. Dies wurde so definiert:

self.softmax = nn.Softmax(dim=1)

Hier sieht man, dass die Dimension noch definiert werden musste, welche hier die erste und nicht die nullte ist, da in der nullten die Batch-Size ist. Die Batch-Size ist nur vorhanden, da ich die Daten zuvor durch einen Datenlader laufen liess, um sie zu sortieren und auch in eine zu laden Grösse umzuformen. Diesen Datenloader werde ich nicht weiter besprechen, da er nicht im Mittelpunkt dieser Arbeit stand. Dies sind die zwei einzigen von mir definierten Funktionen, ab jetzt geht es nur noch darum die Daten zu laden, umzuwandeln und schliesslich durch unser zuvor besprochenes CNN zu geben.

Der ganze kommentierte Code

Hier ist eine genauere Erklärung zu meinem Code. Hier habe ich zu jeder Zeile einen Kommentar verfasst, welchen diese Erklärt:

Hier werden alle Bibliotheken importiert, welche für das Programm benutzt werden.
Dies wird in der ersten Zelle gemacht, damit man sie nicht mehrmals laden muss
Nach jeder Library werde ich noch den Link dazu angeben

```
# Hier werden alle Bibliotheken importiert, welche für das Programm
benutzt werden.
# Dies wird in der ersten Zelle gemacht, damit man sie nicht
mehrmals laden muss
# Nach jeder Library werde ich noch den Link dazu angeben

import numpy as np
#https://numpy.org
import torch
```

```
# https://pytorch.org
import torch.nn as nn
from torch.utils.data import random_split
from torch.utils.data import Dataset
from torch.utils.data import Subset
import torchvision.datasets as datasets
import torchvision.transforms as transforms

import glob
# https://docs.python.org/3/library/glob.html
import os
# https://docs.python.org/3/library/os.html
from sklearn.model_selection import train_test_split
# https://scikit-learn.org/stable/
from collections import Counter
# https://docs.python.org/3/library/collections.html
from itertools import zip_longest
# https://docs.python.org/3/library/itertools.html

# Dies ist die Lernrate, welche wichtig für die "Schrittgrösse des
Programmes" ist
learning_rate= 0.0000001

# Hier ist die Funktion für das Laden der Daten
# Diese Funktion nimmt alle Bilddaten und die Klassen und steckt
diese in einen Datensatz
# Dieser Datensatz hat dann das Label und die Daten in einem
# Diese Daten müssen durch Iterationen durch diesen herausgelesen
werden
class MyDataset(Dataset):
            def __init__(self, images, labels):
                self.images = images
                self.labels = labels

            def __getitem__(self, index):
                image = self.images[index]
                label = self.labels[index]
                return image, label

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

# Definiere das CNN (genauere Erklärung ist weiter unten in dem
Dokument)
class CNN(nn.Module):
    def __init__(self, num_classes):
        super(CNN, self).__init__()

        # Definiere die Convolutional Schichten
        self.conv1 = nn.Conv2d(in_channels=1, out_channels=16,
kernel_size=3, padding=1)
```

```
        self.conv2 = nn.Conv2d(in_channels=16, out_channels=32,
kernel_size=3, padding=1)
        self.conv3 = nn.Conv2d(in_channels=32, out_channels=64,
kernel_size=3, padding=1)

        # Definiere die  max pooling Schichten
        self.pool = nn.MaxPool2d(kernel_size=2, stride=2)

        # Definiere die fully connected Schichten
        self.fc1 = nn.Linear(in_features=576, out_features=128) #64,
128
        self.fc2 = nn.Linear(in_features=128,
out_features=num_classes)

        # Definiere die voll aktiven Funktionen
        self.relu = nn.ReLU()
        self.softmax = nn.Softmax(dim=1)

    def forward(self, x):
        #Alle Funktionen anwenden
        x = self.conv1(x)
        x = self.relu(x)
        x = self.pool(x)
        x = self.conv2(x)
        x = self.relu(x)
        x = self.pool(x)
        x = self.conv3(x)
        x = self.relu(x)
        x = self.pool(x)

        # Den output verflachen
        x = x.view(x.size(0), -1)

        # Die voll-verbunden Schichten anwenden
        x = self.fc1(x)
        x = self.relu(x)
        x = self.fc2(x)

        # Die softmax Funktion anwenden
        x = self.softmax(x)

        return x



# Hier wird der Weg zu den Daten definiert, dieser Variiert mit
jedem Nutzer
# In dem Ordner, den man dort findet hat es noch verschiedene
Unterordner
# In diesen Unterordner sind die Daten zu finden
folder_path =
```

```
r'C:\Users\41793\Desktop\Programming\BG_Creative_Coding\rnn_tutorial
_npydata_subfolder2'

# Hier iterieren wir durch die verschiedenen Ordnern
# Der ganze nachfolgende Code wird pro Ordner einmal ausgeführt
# In meinem Fall hat es in diesem Ordner 35 Unterordner mit jeweils
10 Klassen, mit jeweils ungefähr 150'000 Bildern
# Dies bedeutet, dass hier der untenstehende Code 35 mal abgerufen
wird
for folder in os.listdir(folder_path):

    # Dies ist reine Kosmetik, so dass ich besser sehen kann bei
welchem Ordner wir sind
    # Nach dieser Ausgabe des Ordners in welchem wir und befinden,
folgen dann später die Genauigkeiten des Programmes
    print("*********")
    print(folder)
    print("*********")


    # Hier erhalten wir den kompletten Weg zu dem Ordner in welchem
sich jetzt die 10 Klassen befinden
    file_path = os.path.join(folder_path, folder)

    # Hier sehen wir ob der obige Ordner Weg ein weiterer Ordner hat
    if os.path.isdir(file_path):
        # Falls dies der Fall ist, soll durch diese Daten durch
iteriert werden
        for subfile in os.listdir(file_path):
            # Hier erhalten wir den vollen Weg zu jenem Unterordner
            subfile_path = os.path.join(file_path, subfile)

        # Jetzt sagen wir, dass die Daten welche in diesem Ordner
sind files heissen, solange sie mit .npy enden
        # Hier nutzen wir die glob Library welche es uns ermöglicht
einfacher auf Ordner auf unserem Gerät zuzugreifen
        files = glob.glob(os.path.join(file_path, "*.npy"))

        # Nun setzten wir leere Listen für die Daten und Labels fest
        data_list = []
        label_list = []
        label_list_alone = []
        i = 0
        # Hier gehe wir über die verschiedenen Dateien in Dateien
        for file in files:
            # Nun legen wir das Label fest
            # Da dies noch nicht in den Daten vorgegeben wurden,
nehmen wir einfach den Namen der Datei
            # Dies erreichen wir wieder mit der glob Library, indem
wir einfach nur den Basename nehmen
            # Dieser Basename ist somit der einfache Name der Datei,
```

```
ohne alle Ordner, welche davor noch kämen
            # Wir spalten also den String bei diesem Basename und
nehmen dann nur den Teil in welchem der Name der Datei ist
            label = os.path.splitext(os.path.basename(file))[0]

            # Hier öffnen wir die Daten in Binarymode
            # Wir können die Daten nur mit dem rb = read in binary
öffnen
            # Dies ist der Fall, da man nicht wie normal die Daten
mit r laden kann, wegen ihres Formates
            # Passen sie hier auf, dass sie die Daten nicht mit wb
öffnen, da die Daten sonst automatisch überschrieben werden
            # In unserem Fall wäre dies ungünstig, da wir nichts
reinschreiben und die daten somit leer werden
            with open(file, 'rb') as fin:
                # Nun werden die Daten mit der Library Numpy geladen
                data = np.load(fin)
                # Da wir nur ein Label pro Datei haben müssen wir
dieses vervielfältigen
                # Falls wir dies nicht tun, haben wir das Problem,
dass die Daten-liste und Label-Liste nicht gleich lang ist
                # Wir lösen dies, indem wir einfach für die Länge
der Daten das Label der Label-Liste hinzufügen
                for i in range (0, len(data)):
                    label_list.append(label)
            # Hier fügen wir nun auch die Daten der Daten-Liste
hinzu
            data_list.append(data)

        # Dies ist die Grösse, mit welcher die Daten geladen werden
        batch_size = 100

        # Concatenate data and labels into arrays
        # Nun verketten wir die Daten und Labels zu Reihen, hier
Array genannt
        # Die Daten werden hier auf eine Achse beschränkt, damit sie
die gleichen Dimensionen wie die Labels haben
        data = np.concatenate(data_list, axis=0)
        labels = np.array(label_list)

        # Jetzt machen wir aus den Daten Kommazahlen, da eine
spätere Funktionen nicht mit Integers funktioniert
        data = data.astype(float)

        # Hier machen wir ein Wörterbuch für die Labels
        # Dies wird hier benötigt, da mein CNN nur Zahlen als Input
nehmen kann
        # Das heisst, dass die Labels auch in Zahlenform sein
müssen, damit eine Evaluation stattfinden kann
        # Dieses Wörterbuch hat nun zum Beispiel den Eintrag: "Auto
= 1, Biene = 2, etc."
```

```
        label_dict = {label: i for i, label in
enumerate(set(labels))}

        # Convert labels to integers using the dictionary
        # Nun verwandeln wir die Labels, welche noch Strings sind,
zu Integers
        # Dies erreichen wir, indem wir das label_dict verwenden
        labels = np.array([label_dict[label] for label in labels])

        # Dies ist mehr eine Absicherung, dass auch wirklich alles
eine Array ist
        data = np.array(data)
        labels = np.array(labels)

        # Hier sehen wir definieren wir, ob die Daten farbig oder
schwarz, weiss sind
        # Falls num_channels = 3 sind, dann wäre es farbig, aufgrund
der drei RGB Farben
        # Falls jedoch, wie in unserem Fall dies 1 beträgt sind die
Daten Schwarz-Weiss
        num_channels = data.ndim - 1

        """
        # Hier könnte man noch einige Bilder ausgebe lassen, falls
man eine Idee haben will, wie die Daten aussehen
        import matplotlib.pyplot as plt

        for i in range(1):
            sample = data[i]
            print(f'Sample {i}: {sample}')

        # Wir nehmen an, das die Daten eine 2D Reihe sind, mit Form
(höhe, breite)
            image = np.array(sample)
            image = image.reshape(28, 28)

            # Zeig das Bild
            plt.imshow(image)
            plt.show()
            """

        # Hier sagen wir, das 80% der Batchsize die Input Batchsize
ist
        # Dies wird noch später verwendet, um die Menge der Daten zu
bestimmen
        input_batch = int(0.8* batch_size)

        # Diese Funktion hätte direkt im CNN definiert werden können
        # Ich habe sie hier definiert, da ich es erst während des
schon laufenden Programmes bemerkte
        # Es mach keinen Unterschied wo es definiert wird
```

```
        output_features = 24 * 24

        # Während des Testes hatte ich mehrere Male den Fehler wegen
des Type
        # Wegen dem habe ich die Daten hier noch einmal, ist jedoch
nicht nötig
        # Da dieser Code hier nur 35 Mal ablauft, hat dies keinen
grossen Einfluss auf die Effizienz
        data = np.array(data)
        labels = np.array(labels)

        # Hier würden die Daten auf der CPU geladen werden, falls
die Grafikkarte zu schwach wäre, oder man keine hat
        #data = data.cpu().numpy()
        #labels = labels.cpu().numpy()

        # Create a dataset from your data and labels
        # Nun kreieren wir ein Datenset mit unseren Daten und Labels
        # Man greift auf die zu Beginn definierte Funktion MyDataset
zurück
        dataset = MyDataset(data, labels)

        # Hier definieren wir noch einige notwendige Variablen für
den Rest des Programmes
        input_size = data.shape[1]
        hidden_size = 128
        num_layers = 1
        num_classes = len(label_dict)
        batch_size = batch_size

        # Jetzt wird das CNN initialisiert, das heisst geladen
        # Die num_classes sind die Anzahl an Klassen, welche wir
haben, dies kann aus dem Wörterbuch herausgelesen werden
        cnn = CNN(num_classes=num_classes)

        # Da das CNN in unserem Fall sehr viel Rechenleistung
beanspruch müssen wir es wenn möglich verschieben
        # Dies erreichen wir, indem wir mit torch.device nachschauen
was vorhanden ist
        # Für den Fall, das eine Grafikkarte vorhanden ist und Cuda
unterstützt wird, ist device = GPU
        # Dies ist stark zu empfehlen, da die Rechenzeit sonst viel
zu hoch ist
        # Für den Fall das keine GPU vorhanden ist, wird einfach die
CPU verwendet
        device = torch.device("cuda" if torch.cuda.is_available()
else "cpu")
        # Hier wird das CNN zu dem verfügbaren Gerät verschoben
        cnn.to(device)

        # Nun definieren wir die Loss und Optimizer Funktionen
```

```
        # Diese sind essenziell für das CNN, da es sonst nicht
lernen könnte
        # Die CrossEntropyLoss Funktion ist eine häufig verwendete
Verlust-Funktion
        # Es misst den unterschied zischen der Vorhergesagten
Verteilung der Klassen und der realen Verteilung
        # Dies wird vereinfacht berechnet, in dem man den
unterschied nehmen und diesen dann mit -y * ln(y) multiplizieren
        # Hier wird dies ausführlich erklärt:
https://vitalflux.com/mean-squared-error-vs-cross-entropy-loss-
function/
        criterion = nn.CrossEntropyLoss()
        # Nun haben wir hier den optimizer
        # Hier wurde der Adam optimizer gewählt, welcher einer der
häufigsten ist
        # Die Wahl wurde auf Grund von Tests genommen
        # In einem CNN (Convolutional Neural Network) braucht man
einen Optimierer, um die Gewichte des Netzwerks zu aktualisieren und
zu optimieren. Der Optimierer ist dafür verantwortlich, den Fehler
des Netzwerks zu minimieren, indem er die Gewichte des Netzwerks
anpasst, um die Vorhersagen des Netzwerks so nah wie möglich an den
tatsächlichen Ergebnissen zu bringen.
        # Ein Optimierer verwendet die Gradienten des Verlustes, die
durch die Backpropagation berechnet werden, um die Gewichte des
Netzwerks zu aktualisieren. Der Gradient beschreibt die Änderung des
Verlustes in Bezug auf die Gewichte und die Optimierer nutzen diese
Informationen, um die Gewichte zu aktualisieren und den Verlust zu
minimieren.
        # Der Adam Optimierer kombiniert die Vorteile von zwei
anderen Optimierern: dem Momentum Optimierer und dem AdaGrad
Optimierer.
        # Der Momentum Optimierer hilft dabei, das Netzwerk
schneller aus flachen Minima herauszubewegen und das AdaGrad
Optimierer hilft dabei, die Lernrate für jeden Parameter anzupassen.
        # Adam nutzt diese zwei Ansätze und ist somit populär
        # Für eine genauere Erklärung verweise ich gerne auf diesen
Artikel: https://artemoppermann.com/de/optimierung-in-deep-learning-
adagrad-rmsprop-adam/
        optimizer = torch.optim.Adam(cnn.parameters(), lr =
learning_rate)

        test_batch = int(batch_size * 0.2)

        # Nun laden wir die Daten mit einer der vielen Funktionen
von PyTorch
        # Diese Funktion erstellt einen Dataloader, der die Daten in
Batches aufteilt und sie zufällig mischt (shuffle=True) und das
letzte Batch verwirft (drop_last=True), wenn es nicht vollständig
gefüllt ist.
        dataloader = torch.utils.data.DataLoader(dataset,
batch_size, shuffle=True, drop_last=True)
```

```
        # Hier teile ich die Daten in Training und Test Daten
        train_set_size = int(len(dataset) * 0.8)
        test_set_size = len(dataset) - train_set_size

        # Nun definieren wir die Anzahl an Epochen für welches das
CNN trainieren wird
        # Ich hatte diesen Wert meistens bei 125 um das Netz zu
trainieren
        # Dies führte dazu, dass ich meinen PC  für eine ganze Woche
laufen lies
        # Die Trainingszweit betrug 150 Stunden, wobei die Resultate
mit längerem lernen noch besser werden würden
        num_epochs = 150
        # Hier folgt eine for Schleife, welche für die Anzahl an
Epochen ausgeführt wird
        for epoch in range(num_epochs):
            # Jetzt Iterieren wir über den Dataloader, der die Daten
in Batches aufteilt.
            for data, labels in dataloader:
                # Wir wandeln die Labels in den Datentyp Long Tensor
                labels = labels.type(torch.LongTensor)
                # Hier verschieben wir die Daten dann auf die GPU
durch die Funktion .cuda()
                data, labels = data.cuda(), labels.cuda()
                # Wir splitten dann die Daten in Trainings- und
Testdaten mit dem train_test_split() von sklearn
                train_data, test_data, train_labels, test_labels =
train_test_split(data, labels, test_size=0.2)
                # Wir wandeln die Trainingsdaten in den Datentyp
float um ändern die Form der Daten von (batch_size, 28, 28) in
(batch_size, 1, 28, 28)
                train_data = train_data.to(dtype=torch.float)
                # Danach ändern wir die Form der Daten von
(batch_size, 28, 28) in (batch_size, 1, 28, 28)
                train_data = train_data.view(-1, 28, 28)
                train_data = train_data.view(input_batch, 1, 28, 28)
                # Wir führen die Daten durch das CNN und speichern
das Ergebnis in der Variablen "output".
                # Dies ist der Schritt, welcher bei weitem am
meisten Fehler verursacht hatte
                # Dies ist so, da der Code bis hier hin ohne Fehler
gelaufen ist, jedoch dann immer ein Fehler in Bezug auf die Form
hervor kam
                # Dieser Schritt braucht als einzige Variabel
wahrscheinlich am meisten Zeit
                # Zu Grunde liegen die vielen Umwandlung welche in
unserem CNN geschehen
                output = cnn(train_data)
                # Wir berechnen den Verlust (loss) mit der
verwendeten Loss-Funktion (criterion) zwischen dem output des Netzes
```

```
und den tatsächlichen Trainingslabels.
                # Dies geschieht einfach indem wir die zuvor
definierte Funktion aufrufen
                loss = criterion(output, train_labels)
                # Wir setzen die Gradienten auf null (zero_grad())
                optimizer.zero_grad()
                # Nun führen einen Backpropagation Schritt aus
(backward())
                loss.backward()
                # Zu Letzt aktualisieren wir die Gewichte des Netzes
mit dem verwendeten Optimierer (optimizer.step()).
                optimizer.step()

            # Wenn die aktuelle Epoche ein Vielfaches von 1 ist,
geben wir den aktuellen Verlust aus.
            # Mit den Werten welche ich gewählt hatte, passiert dies
alle 100 Sekunden
            if (epoch+1) % 1 == 0:
                print(f'Epoch [{epoch+1}/{num_epochs}], Loss:
{loss.item():.4f}')


        # Dieser Code stellt eine Schleife dar, die verwendet wird,
um die Genauigkeit des trainierten CNNs auf einem Testdatensatz zu
überprüfen
        # Nun setzen wir das CNN zu Evaluations-Modus, dies
bedeutet, dass das CNN nicht trainiert wird
        cnn.eval()
        # Hier schalten wir die gradient Optimierung ab
        with torch.no_grad():
            # Nun deklarieren wir die Variablen correct und total
das erste Mal
            # Wir setzen diese auf null, da dies der natürliche
Ausgangspunkt ist
            correct = 0
            total = 0
            # Nun iterieren wir genau gleich wie oben durch die
Daten und verwenden die Testdaten
            # Hier trainieren wird das Model jedoch nicht, sondern
schauen einfach, was die Gewichte ausgeben würden
            for data, labels in dataloader:
                labels = labels.type(torch.LongTensor)
                data, labels = data.cuda(), labels.cuda()
                train_data, test_data, train_labels, test_labels =
train_test_split(data, labels, test_size=0.2)
                test_data = test_data.to(dtype=torch.float)
                test_data = test_data.view(-1, 28, 28)
                test_data = test_data.view(test_batch, 1, 28, 28)
                # Hier werden die Daten wieder durch das CNN gegeben
und es gibt uns somit die Vorhersage zurück
                outputs = cnn(test_data)
```

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

                # Berechne die Anzahl an richtigen Vorhersagen
                correct = (predicted == test_labels).sum().item()

                # Nun berechnen wir in Prozent wie genau unser Model
ist
                accuracy = 100 * correct / test_labels.size(0)
            # Nun geben wir die Genauigkeit aus.
            # Dies geschieht am Ende jedes Ordners und dies ist auch
die Genauigkeit mit welcher das Model neue Bilder richtig zuordnen
wird
            # Diese Genauigkeit beträgt bei meinem Training im
Schnitt um die 75%
            print("The Accuracy of the CNN is:", accuracy)

        # Damit wir dieses Model wieder verwenden können, müssen wir
die Gewichte des Models in einem .pt speichern
        # Um zu verhindern, dass wir die schon zu vor gespeicherten
Werte überschreiben, nehmen wir die ändernde Variable folder in den
Namen der Datei
        # Durch diesen Namen ist es auch einfacher wieder
sicherzustellen zu welchem Subfolder die Gewichte gehören
        torch.save(cnn.state_dict(), 'cnn_{}.pt'.format(folder))

        # Mir steht nur eine gewisse Anzahl an Speicher zur
Verfügung
        # Auf Grund dessen, muss ich die Variabel data löschen
        # Diese Variabel beansprucht in meinem Fall 10 GB an RAM und
auch noch alle 8 GB VRAM meiner GPU
        del data
```

Weitere Codes

Ich habe noch viele weitere Programme geschrieben, als das oben kommentierte. Ich werde nicht alle hier einfügen, jedoch sind sie in einem mitgelieferten Ordner verfügbar und ich würde mich freuen, wenn sie diese Programme bei der Evaluation ihrer Note, insbesondere der Entwicklungsnote Miteinbeziehung. Hier würde ich gerne noch die simplen Programme, welche für das finale Produkt verwendet wurden, einfügen.

Code für das Erstellen der Unterordner

```
import os
import shutil

# Hier nehmen wir den Ort der Dateien
directory =
'C:/Users/41793/Desktop/Programming/BG_Creative_Coding/rnn_tutorial_
npydata_subfolder/'

# Setzt die anzahl an Dateien pro Unterordner
batch_size = 10
```

```
# Eine Liste aller Dateien in diesem Ordner
files = os.listdir(directory)

# Initialisierung für den jetztigen Unterordner
subdir_index = 0

# Einen Unterordner für den ersten "Batch" von Dateien
subdir_name = files[0][:2] + '_' + files[batch_size-1][:2]
subdir_path = os.path.join(directory, subdir_name)
os.mkdir(subdir_path)

# Durch alle Dateien iterieren
for i in range(len(files)):
    # Die aktuelle Datei in den Unterorder verschieben
    file_path = os.path.join(directory, files[i])
    shutil.move(file_path, subdir_path)

    # Wenn wir die Batchgrösse erreicht ahebn, soll ein neuer
Unterordner entstehen
    if (i+1) % batch_size == 0 and i+1 < len(files):
        subdir_index += 1
        subdir_name = files[i+1][:2] + '_' + files[i+1+batch_size-
1][:2]
        subdir_path = os.path.join(directory, subdir_name)
        os.mkdir(subdir_path)
```

Code für das Zeichnen mit dem CNN

Dieser Code ist mein Versuch eigene Zeichnungen zu klassifizieren. Das Programm ist kommentiert, jedoch nicht ganz so ausführlich wie zuvor. Wiederholte Funktionen werden nicht mehrmals erklärt werden. Falls sie dieses Programm nutzen wollen, müssen sie über die Nutzung des Kreuzes am oben rechten Rand auf die nächste Seite gelangen.

```
import pygame
import numpy as np
import torch
import torch.nn as nn
from torch.utils.data import random_split
from torch.utils.data import Dataset
from torch.utils.data import Subset
import torchvision.datasets as datasets
import torchvision.transforms as transforms
from PIL import Image
import cv2
import cairocffi as cairo



import glob
import os
```

```
from sklearn.model_selection import train_test_split
from collections import Counter
from itertools import zip_longest

# Diese zwei funktionen müssen definiert werden, damit das Programm
laufen kann
# Die Funktionen hier wurden jedoch schon zuvor erklärt
class MyDataset(Dataset):
            def __init__(self, images, labels):
                self.images = images
                self.labels = labels

            def __getitem__(self, index):
                image = self.images[index]
                label = self.labels[index]
                return image, label

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

# Define the CNN model
class CNN(nn.Module):
    def __init__(self, num_classes):
        super(CNN, self).__init__()

        # Define the convolutional layers
        self.conv1 = nn.Conv2d(in_channels=1, out_channels=16,
kernel_size=3, padding=1)
        self.conv2 = nn.Conv2d(in_channels=16, out_channels=32,
kernel_size=3, padding=1)
        self.conv3 = nn.Conv2d(in_channels=32, out_channels=64,
kernel_size=3, padding=1)

        # Define the max pooling layer
        self.pool = nn.MaxPool2d(kernel_size=2, stride=2)

        # Define the fully connected layers
        self.fc1 = nn.Linear(in_features=576, out_features=128) #64,
128
        self.fc2 = nn.Linear(in_features=128,
out_features=num_classes)

        # Define the activation functions
        self.relu = nn.ReLU()
        self.softmax = nn.Softmax(dim=1)

    def forward(self, x):
        # Apply the convolutional layers
        x = self.conv1(x)
        #print(x.shape)
        x = self.relu(x)
```

```
        #print(x.shape)
        x = self.pool(x)
        #print(x.shape)
        x = self.conv2(x)
        #print(x.shape)
        x = self.relu(x)
        #print(x.shape)
        x = self.pool(x)
        #print(x.shape)
        x = self.conv3(x)
        #print(x.shape)
        x = self.relu(x)
        #print(x.shape)
        x = self.pool(x)
        #print(x.shape)

        # Flatten the output of the convolutional layers
        x = x.view(x.size(0), -1)
        #print(x.shape)

        # Apply the fully connected layers
        x = self.fc1(x)
        #print(x.shape)
        x = self.relu(x)
        #print(x.shape)
        x = self.fc2(x)
        #print(x.shape)

        # Apply the softmax function
        x = self.softmax(x)
        #print(x.shape)

        return x

cnn = CNN(num_classes = 10)

# Initialisiere Pygame
pygame.init()

# Window grösse festlegen
window_size = (800, 600)

# Ein Window erstellen
window = pygame.display.set_mode(window_size)

# Diesem Window einen Titel geben
pygame.display.set_caption("Drawing Window for QuickDraw")

# Den Font für den Text auswählen
font = pygame.font.Font(None, 30)
```

```
# Mein zuvor kreiirtes Text-Dokument (Code im Ordner) öffnen
with open("categories.txt") as f:
    text = f.read()

# Den Text darstellen
text_surface = font.render(text, True, (255, 255, 255))

# Hintergrund Farbe festlegen
window.fill((255, 255, 255))

# Den Text auf Linien aufteilen
lines = text.split("\n")

# Die Startposition festlegen
scroll_pos = 0

# Eine Verzögerung für das Scrollen festlegen
delay = 50

# Haupt Schleife
running = True
while running:
    for event in pygame.event.get():
        # Hier wird das Programm abgebrochen, für den Fall, das Quit
geklickt wird
        if event.type == pygame.QUIT:
            running = False
    # Das scrollen wird hier beschrieben
    keys = pygame.key.get_pressed()
    if keys[pygame.K_UP]:
        scroll_pos -= 1
    if keys[pygame.K_DOWN]:
        scroll_pos += 1

    # Den Bildschirm löschen
    window.fill((255, 255, 255))

    # Den Text anzeigen
    for i, line in enumerate(lines[scroll_pos:]):
        text_surface = font.render(line, True, (0, 0, 0))
        window.blit(text_surface, (50, 50 + (i * 30)))

    # Den Bildschirm Updaten
    pygame.display.flip()

    pygame.time.wait(delay)

# Pygame schliessen
pygame.quit()

pygame.init()
```

```
window_size = (800, 600)
screen = pygame.display.set_mode(window_size)
pygame.display.set_caption("User Input Example")
font = pygame.font.Font(None, 30)

# Variable um den Text des Users zu speichern
user_text = ""

running = True
while running:
    for event in pygame.event.get():
        if event.type == pygame.QUIT:
            running = False
        elif event.type == pygame.KEYDOWN:
            # Nutzereingaben nutzen
            if event.unicode.isprintable():
                user_text += event.unicode
            elif event.key == pygame.K_BACKSPACE:
                user_text = user_text[:-1]

    screen.fill((255, 255, 255))

    # Den Text Anzeigen
    prompt_text = "Please enter your wished category here:"
    prompt_surface = font.render(prompt_text, True, (0, 0, 0))
    screen.blit(prompt_surface, (50, 50))

    # Das Textfeld anzeigen
    text_field_surface = font.render(user_text, True, (0, 0, 0))
    screen.blit(text_field_surface, (50, 100))

    pygame.display.flip()

# Die Eingabe drucken
print(user_text)

pygame.quit()

pygame.init()
window_size = (800, 600)
window = pygame.display.set_mode(window_size)
pygame.display.set_caption("Drawing Window for QuickDraw")
font = pygame.font.Font(None, 30)
text_surface = font.render(text, True, (255, 255, 255))
window.fill((255, 255, 255))

running = True
drawing = False
while running:
    for event in pygame.event.get():
        # Für den Fall des schliessen des Programmes
```

```
        if event.type == pygame.QUIT:
            running = False

        # Umgang mit Maustaste unten (beginn zu zeichnen)
        elif event.type == pygame.MOUSEBUTTONDOWN:
            drawing = True
            last_pos = event.pos

        # Umgang mit Maustaste Oben (Stop das Zeichnen)
        elif event.type == pygame.MOUSEBUTTONUP:
            drawing = False

        # Umgang mit Mausbewegungen (Linie zeichnen)
        elif event.type == pygame.MOUSEMOTION:
            if drawing:
                pygame.draw.line(window, (0, 0, 0), last_pos,
event.pos, 5)
                last_pos = event.pos

    # Gedruckte Tasten
    keys = pygame.key.get_pressed()
    if keys[pygame.K_s]:
        # Das Bild als JPG speichern
        rgb_image = window.convert()
        pygame.image.save(rgb_image, "drawing.jpg")

    # Der Bildschirm updaten
    pygame.display.update()

pygame.quit()

# Um richtige Klassifikation zu garantieren, müsste ich eine
möglichkeit haben mein JPG durch diese Funktion zu geben
# Diese Funktion würde das Bild in ein, für das CNN vertrautes Bild
verwandeln
# Die orginale Eingabe für diesen Code wären .bit files, welche
Striche als Vektor gespeichert haben
# Mit diesem Schritt hier hatte ich viele schwierigkeiten, jedoch
war es schlussendlich einfach die Zeit welche fehlte
# Die Funktion vector_to_raster wurde auch verwendet, um die
orginalen Daten herzustellen und ist auch auf der Datenseite zu
finden
"""# Load the image from .npy file
img = np.load("drawing.npy")

print(img.shape)
# if necessary Convert to grayscale
#img = cv2.cvtColor(img, cv2.COLOR_RGB2GRAY)
img = img.astype(np.uint8)

# Apply Gaussian Blur to reduce noise
```

```
img = cv2.GaussianBlur(img, (5,5), 0)

# Apply Canny Edge Detection
edges = cv2.Canny(img, 50, 150)

# Find lines using HoughLinesP
lines = cv2.HoughLinesP(edges, 1, np.pi/180, 50, minLineLength=50,
maxLineGap=5)

# Extract x,y coordinates of the lines
lines_coordinates = [[(line[0][0], line[0][1]), (line[0][2],
line[0][3])] for line in lines]


def vector_to_raster(vector_images, side=28, line_diameter=16,
padding=16, bg_color=(0,0,0), fg_color=(1,1,1)): #vector_images

    padding and line_diameter are relative to the original 256x256
image.


    original_side = 256.

    surface = cairo.ImageSurface(cairo.FORMAT_ARGB32, side, side)
    ctx = cairo.Context(surface)
    ctx.set_antialias(cairo.ANTIALIAS_BEST)
    ctx.set_line_cap(cairo.LINE_CAP_ROUND)
    ctx.set_line_join(cairo.LINE_JOIN_ROUND)
    ctx.set_line_width(line_diameter)

    # scale to match the new size
    # add padding at the edges for the line_diameter
    # and add additional padding to account for antialiasing
    total_padding = padding * 2. + line_diameter
    new_scale = float(side) / float(original_side + total_padding)
    ctx.scale(new_scale, new_scale)
    ctx.translate(total_padding / 2., total_padding / 2.)

    raster_images = []
    for vector_image in vector_images:
        vector_image = np.array(vector_image)
        print(type(vector_image))
        print(vector_image.shape)
        # clear background
        ctx.set_source_rgb(*bg_color)
        ctx.paint()
        #vector_image = np.concatenate(vector_image)

        bbox = np.hstack(vector_image).max(axis=0)
        #bbox = np.concatenate(vector_image)
        print(bbox.shape)
```

```
        print(bbox)
        offset = ((original_side, original_side) - bbox) / 2.
        offset = offset.reshape(-1,1)
        centered = [stroke + offset for stroke in vector_image]

        # draw strokes, this is the most cpu-intensive part
        ctx.set_source_rgb(*fg_color)
        for xv, yv in centered:
            ctx.move_to(xv[0], yv[0])
            for x, y in zip(xv, yv):
                ctx.line_to(x, y)
            ctx.stroke()

        data = surface.get_data()
        raster_image = np.copy(np.asarray(data)[::4])
        raster_images.append(raster_image)

    return raster_images

data = vector_to_raster(vector_images= lines_coordinates)
"""
# Dies ist der Weg zu dem vorher gespeicherten Bild
path = 'drawing.jpg'
# Das Bild öffnen
image = Image.open(path).convert('L')
# Hier könnte man sich das orginal anzeigen lassen
#image.show()
# Das Bild verkleinern
image = image.resize((28, 28))
# Hier könnte man sich das verwandelte Bild anzeigen lassen
#image.show()
# Hier speichern wir das verkleinerte Bild als JPG und als NPY
image.save('drawing_28x28.jpg')
np.save('drawing.npy', image)

# Hier öffnen wir die verschiedenen Textdateien mit den Labels der
Daten
# Diese .txt Daten wurden auch mit einem Code in dem Ordner
generiert und gespeichert
# Die Datei wird aufgrund der zuvorigen Nutzereingabe ausgewählt
with open("Categories_Text/{}.txt".format(user_text), "r") as f:
    class_labels = [line.strip() for line in f.readlines()]


# Hier laden wir unser CNN und schieben es auf die CPU
# Dies tun wir, da es nur ein Bild ist und man dies nicht auf einer
GPU berechnen muss
# Diese Datei wird auch durch die vorherige Nutzereingabe bestimmt
state_dict = torch.load('cnn_{}.pt'.format(user_text),
map_location='cpu')
```

```
# Denn Model Status festlegen
cnn.load_state_dict(state_dict)
# Das model von training zu evaltuation setzen
cnn.eval()

#Die Daten laden
data = np.load("drawing.npy")
# Das Bild zu einem Tensor verwandeln
input_tensor = torch.from_numpy(data)
# Die Daten zu float verwandeln
input_tensor = input_tensor.float()

# Den Tensor umformen um den erwarteten Eingaben des Models zu
entsprechen
input_tensor = input_tensor.reshape((1, 1, 28, 28))

# Ohne das Model zu trainieren, wird das Bild durch das Model
gebracht
with torch.no_grad():
    output = cnn(input_tensor)

# Nun entnehmen wir dem Model die Klasse mit der höchsten
Wahrscheinlichkeit
_, predicted = torch.max(output.data, 1)

# Jetzt nehmen wir das Label der vorhergesagten Klasse
predicted_class_label = predicted.item()

# Nun drucken wir die Klasse und das Label
print("Predicted class label:", predicted_class_label)
print("Predicted class name:", class_labels[predicted_class_label])


pygame.init()
size = (800, 600)
screen = pygame.display.set_mode(size)
screen.fill((255, 255, 255))
pygame.display.set_caption("Prediction Result")

# Den Text anzeigen
font = pygame.font.Font(None, 32)
text = font.render("Predicted class label:
{}".format(predicted_class_label), True, (0, 0, 0))
text2 = font.render("Predicted class name:
{}".format(class_labels[predicted_class_label]), True, (0, 0, 0))
screen.blit(text, (50, 100))
screen.blit(text2, (50, 150))
pygame.display.flip()

# Die Pygame Schlaufe
running = True
```

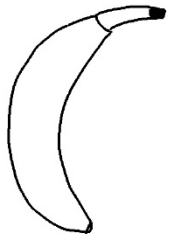
```
while running:
    for event in pygame.event.get():
        if event.type == pygame.QUIT:
            running = False

# Exit Pygame
pygame.quit()
```

Resultate dieses Programmes

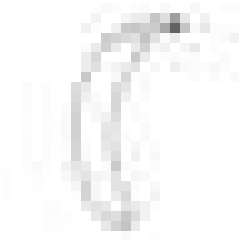
Dieses Programm hat das oben angemerkte Problem, dass die Daten nicht richtig verarbeitet werden können. Dies hat zur Folge, dass das Programm eine niedrigere Genauigkeit hat. Der Grund hierfür ist, dass die Daten verschwommen sind, da sie in meinem Fall, auf unvorsichtige Art, von 800 mal

600 auf 28 mal 28 Pixel reduziert werden. Dies kann halbwegs umgangen werden, indem man eine sehr simple Zeichnung macht. Hier ein Beispiel:

Eine klare Zeichung
Das Bild einer Banane
(800 mal 600)



Eine unscharfe Zeichnung
Die Umrisse einer Banane
(28 mal 28)



In diesem Beispiel sieht man, dass die Zeichnung schwierig zu erkennen wird, da bei der Verkleinerung nicht auf das Erhalten scharfer Kanten geachtet wurde. Da diese Zeichnung jedoch sehr simpel ist, konnte das CNN dieses Bild richtig zuordnen und lieferte diese Ausgabe:

```
Predicted class label: 3
Predicted class name: banana.npy
```

Dies zeigt uns, dass das Programm in seinen grobsten Zügen funktioniert, jedoch muss die Bildverarbeitung noch verbessert werden soll.

Alle verwendeten Links

Da ich volle Transparenz will und sie auch sehen sollen, wie viel ich an diesem Projekt gearbeitet und auch gewachsen bin, habe ich alle meine verwendeten Links in der Zeit des Erschaffens dieses Projektes in zwei Dateien gepackt. Die eine Datei ist von meinem Laptop, die andere von meinem PC zuhause. Diese Dateien umfassen ungefähr einhundert Seiten, wobei zu jedem Link auch noch die Uhrzeit des Aufrufens aufgelistet ist.