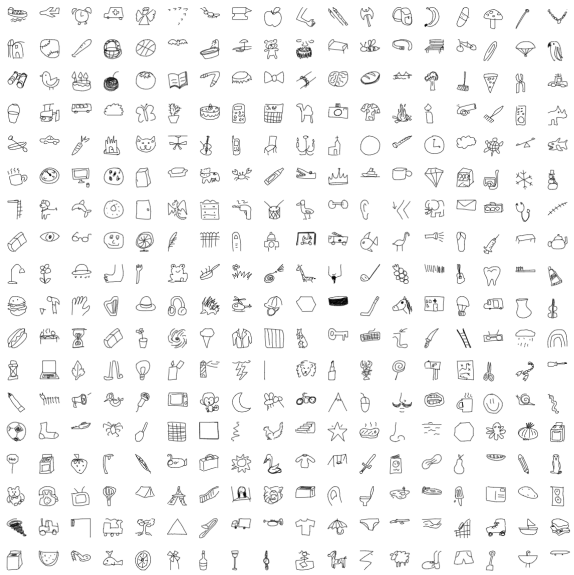
Klassifizierung von handgezeichneten Gegenständen mittels künstlicher Intelligenz

Maturaarbeit
Kantonsschule am Burggraben St. Gallen



Vorgelegt durch: Eric Ceglie
Vorgelegt bei: Dr. Simon Knaus

Datum des Einreichens: 31. Januar 2020

KANTONSSCHULE AM BURGGRABEN

Inhaltsverzeichnis

1 Einleitung

Maschinelles Lernen ist heute so präsent wie nie zuvor. Während das Konzept der künstlichen Intelligenz erst im Sommer 1956 mit der Dartmouth Conference [16] offiziell als akademisches Fachgebiet anerkannt wurde und vor 20 Jahren noch mit massiven Imageproblemen zu kämpfen hatte, ist es heute nicht mehr aus unserem Alltag wegzudenken. Nicht nur für soziale Medien, für wirtschaftliche Marktanalysen, für die Medizin oder für unsere Smartphones ist maschinelles Lernen von hoher Bedeutung, sondern auch Fahrzeuge sind schon fast in der Lage völlig autonom im herkömmlichen Strassenverkehr zu manövrieren. Unter den verschiedene Methoden des maschinellen Lernens wurden künstliche neuronale Netze nach dem biologischen Vorbild des Gehirns über die letzten Jahre immer bekannter und erfolgreicher. Zu diesem Aufschwung haben bessere Computer und vor allem die Verfügbarkeit von riesigen Datenmengen beigetragen.

Es gab verschiedene Gründe, die mich zu meiner Themenwahl führten. Zum einen konnte ich mich aufgrund der Aktualität dieses Themas schon länger für das maschinelle Lernen im Allgemeinen begeistern, doch was mich stets am meisten interessierte, war die Technik hinter den Maschinen. Es waren vor allem die theoretischen Konzepte der Mathematik und Informatik, die meine Aufmerksamkeit erregten und da ich schon seit Langem wissen wollte, wie maschinelles Lernen mit künstlichen neuronalen Netzen funktioniert, habe ich mich dazu entschieden, die Maturaarbeit als Chance dafür zu nutzen. Zudem war mir schon vor der Themenwahl bewusst, dass ich neben der schriftlichen Arbeit auch gerne ein Produkt erstellen würde. Als ich dann begann die Programmiersprache Python kennenzulernen, führte dies schnell zur endgültigen Entscheidung eine künstliche Intelligenz zu programmieren.

Das Hauptziel meiner Arbeit war das Erstellen eines Programms mit der Programmiersprache Python zur Klassifizierung von handgezeichneten Gegenständen mit einer künstlichen Intelligenz, wobei mich unter anderem der riesigen *Quick, Draw!* Datensatz von Google inspirierte. Um ein möglichst effektives Programm zu erstellen, habe ich mich zwar dazu entschieden, die äusserst bekannte Programmbibliothek *PyTorch* zu verwenden, doch trotzdem war mein zusätzliches Ziel die grundlegende Mathematik des Trainings eines einfachen neuronalen Netzes mit dem Gradientenabstiegsverfahren als Optimierungsalgorithmus zu verstehen.

Zuerst wird in Kapitel 2 meiner Maturaarbeit die Struktur und der Umgang mit dem *Quick, Draw!* Datensatz erläutert. In Kapitel 3 wird dann ein einfaches neuronales Netz mathematisch beschrieben und es werden die unter anderem auch von *PyTorch* verwendeten Gleichungen für das Gradientenabstiegsverfahren hergeleitet. In Kapitel 4 werden verschiedenste, während meiner praktischen Arbeit verwendete, Möglichkeiten zur Optimierung des Trainings eines neuronalen Netzes vorgestellt. Zum Schluss werden in Kapitel 5 die Ergebnisse meiner praktischen Arbeit zusammengetragen, vorgestellt und analysiert. Zudem befinden sich im Anhang die Quellcodes zu den vier von mir erstellten und verwendeten Programmen, die einen wesentlichen Teil meiner Maturaarbeit ausmachen. Deshalb wurden der Abgabe der schriftlichen Arbeit auch Kopien der vier Programme beigelegt.

2 Der Quick, Draw! Datensatz

Quick, Draw! [13] (kurz: QD) ist ein Onlinespiel, welches von Google, insbesondere für Forschungszwecke, erstellt wurde. Dabei soll man einen vorgegebenen Begriff entweder solange zeichnen, bis ihn der Computer mit Hilfe von einer künstlichen Intelligenz errät, oder bis die gegebene Zeit abläuft. Oft erkennt die Maschine den gezeichneten Gegenstand sogar schneller, als man es von einem Menschen erwarten könnte, während doch sehr selten auch der Fall zustande kommen kann, dass für unser Auge offensichtliche Zeichnungen von dem Computer nicht erfolgreich klassifiziert werden können.

Dieses Spiel ist schon seit November 2016 für jeden im Internet frei verfügbar und es ist auch durchaus bekannt geworden, wodurch Google über die Jahre äusserst viele Daten von menschlich gezeichneten Gegenständen sammeln konnte. Weil dieses Projekt von Anfang an der Forschung dienen sollte, ist der riesige QD Datensatz [1] frei zugänglich und wird auch ziemlich strukturiert in verschiedenen Formaten bereitgestellt. Mit über 50 Millionen Bildern, die in 345 Kategorien eingeteilt sind, gehört der QD Datensatz zu den grössten öffentlich zugänglichen und strukturierten Datensammlungen. Im folgenden Kapitel wird der Aufbau des Datensatzes genauer erläutert.

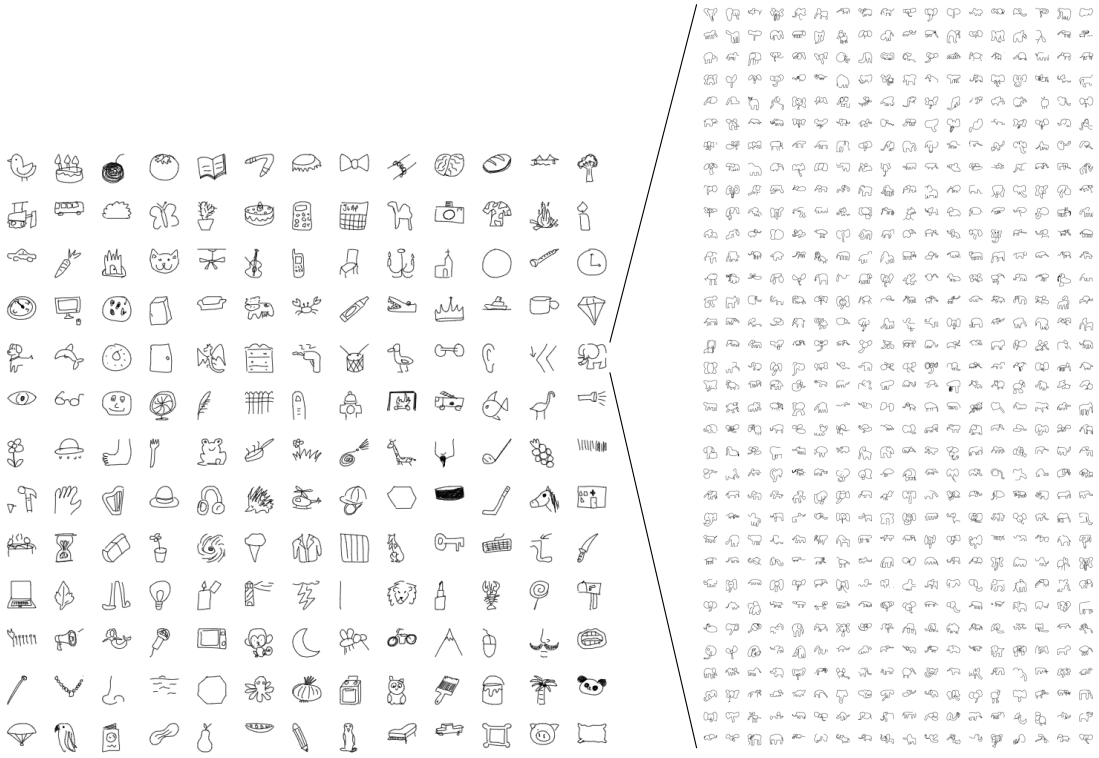


Abbildung 1: Ausschnitt aus dem Quick, Draw! Datensatz

Quelle: `https://quickdraw.withgoogle.com/data`

2.1 Aufbau der Daten

Google stellt die Daten in drei verschiedenen Formaten zur Verfügung:

Numpy Rastergrafik (.npy): Dieses Format wäre eigentlich optimal für meine Arbeit gewesen, wenn die einzelnen Bilder nicht schon auf 28×28 Pixel reduziert wären. Da Rastergrafik Dateien jeden Pixel eines Bildes speichern, werden diese schnell extrem gross. Deshalb hat sich Google dazu entschlossen bei den .npy Dateien die Auflösung zu reduzieren. Doch die originalen 256×256 Bilder werden in anderen, ressourceneffizienteren Formaten zur Verfügung gestellt.

Newline-delimited JSON (.ndjson): Newline-delimited JSON [12] (kurz: NDJSON) ist eine Variante des JSON streaming Verfahrens, was wiederum eine Erweiterung des klassischen JavaScript Object Notation (kurz: JSON) Datenformats bildet. Da die Verarbeitung der Daten durch Programme bei NDJSON, anders als bei herkömmlichen JSON Daten, zeilenweise geschieht, sind diese grundsätzlich flexibler. In diesem Format werden die originalen 256×256 Bilder abgespeichert und dies geschieht auch ziemlich ressourceneffizient, was später im Code 2 noch genauer ersichtlich wird. Aufgrund der hohen Flexibilität und Kompatibilität des NDJSON Formats mit Python und aufgrund der Erhaltung der ursprünglichen Bildgrösse, habe ich während meiner Arbeit dieses Format verwendet.

Binärdateien (.bin): Zusätzlich gäbe es noch das bezüglich des Ladens der Daten etwas effizientere Format der Binärdateien, doch aufgrund von persönlichen Präferenzen habe ich mich für das NDJSON Format entschieden.

```
{
  "key_id": "5891796615823360",
  "word": "apple",
  "countrycode": "AE",
  "timestamp": "2017-03-01 20:41:36.70725 UTC",
  "recognized": True,
  "drawing": [[[129, 128, 129, 129, 130, 130, 131, 132, 132, ...], ...], ...]
}
```

Code 1: Beispiel einer Dateneinheit des QD-Datensatzes

Code 1 zeigt einen exemplarischen Ausschnitt des QD-Datensatzes. Hier wird eine mögliche Dateneinheit der apple.ndjson Datei gezeigt. Wird diese Datei mit Python geladen, so kann das Programm dann mit verschiedenen Schlüsselwörtern gezielt auf die gewollten Inhalte von bestimmten Dateneinheit zugreifen. Mögliche Schlüsselwörter sind:

- **key_id:** liefert eine für jede Dateneinheit einzigartige Zahl
- **word:** liefert den Namen des zugehörigen Gegenstandes
- **countrycode:** liefert den Standort der Entstehung des Bildes
- **timestamp:** liefert den Zeitpunkt der Entstehung des Bildes

- **recognized:** gibt an, ob der gezeichnete Gegenstand von Googles KI richtig erkannt wurde
- **drawing:** liefert den Bildarray des zugehörigen Gegenstandes

Dabei sind *word*, *drawing* und *recognized* die einzig relevanten Schlüsselwörter für meine Arbeit. Mit dem Schlüssel *drawing* werden die benötigten Informationen zur Rekonstruktion des Bildes extrahiert. Danach kann die entstandene Rastergrafik mit dem Schlüssel *word* dem entsprechenden Gegenstand zugeordnet werden, was erst ein überwachtes Lernen der KI ermöglicht. Das Schlüsselwort *recognized* wäre zwar nicht zwingend notwendig, doch damit kann die Qualität der Trainingsdaten erhöht werden, was durchaus zu einem besseren Ergebnis meiner KI beitragen kann.

```
[
  [   # erster Pinselstrich
    [x0, x1, x2, x3, ...],
    [y0, y1, y2, y3, ...],
    [t0, t1, t2, t3, ...]
  ],
  [   # zweiter Pinselstrich
    [x0, x1, x2, x3, ...],
    [y0, y1, y2, y3, ...],
    [t0, t1, t2, t3, ...]
  ],
  ... # zusaetzliche Pinselstriche
]
```

Code 2: Beispiel eines Bildarrays aus dem QD-Datensatz

Werfen wir nun einen genaueren Blick auf die Struktur der zur Verfügung gestellten Bildarrays, wobei Code 2 wiederum einen exemplarischen Ausschnitt repräsentiert. Vorab wäre aber noch erwähnenswert, dass die .ndjson Daten sogar deutlich weniger Speicherkapazität einnehmen, als die .npy Rastergrafik Daten und dabei trotzdem viel mehr Informationen enthalten, wodurch sie eine um ein vielfaches höhere Bildqualität bieten können, wenn man die Daten nur richtig kompiliert.

Ein Bild wird jeweils in einzelne Pinselstriche gegliedert, wobei ein Pinselstrich in dem Spiel mit dem Drücken der linken Maustaste begonnen wird und solange anhält, bis die Maustaste wieder losgelassen wird. Dabei werden während jedem Pinselstrich in bestimmten Zeitabschnitten Δt die momentane Position x_i und y_i, sowie die momentane, für meine Arbeit eher uninteressante, Zeit t_i festgehalten. Dieses Konzept ermöglicht es viele Informationen relativ kompakt (vor allem im Vergleich zur den eben genannten Rasterdateien) zu speichern. Jedoch wird dadurch zugleich die Arbeit mit den Daten etwas komplizierter.

Für meine Arbeit musste ich die wichtigsten Informationen aus dem Datensatz extrahieren und die einzelnen Bilder in eine beliebige n×n Rastergrafik umwandeln. Zudem habe ich ein eigenes System zur richtigen Beschriftung der Bilder erarbeitet, welches nicht nur mit Strings, sondern auch mit Zahlen funktioniert, damit es mit der Ausgabe eines neuronalen Netzes kom-

patibel ist. In den folgenden Kapiteln wird das von mir erstellte Verfahren genauer erläutert. Der komplette Quellcode für meinen Compiler befindet sich im Anhang B.

2.2 Umformen der Daten

Bei der Zusammenstellung von einem Datensatz müssen im ersten Schritt die von Google bereitgestellten Informationen in einheitliche Rastergrafiken eingebettet werden. Komfortabler Weise habe ich mich bei den Bildern für die Hintergrundfarbe Schwarz entschieden, da PyTorch eine Funktion hat, die es erlaubt einen Array beliebiger Form gleich beim Erstellen mit lauter Nullen zu füllen.

Somit wird zuerst ein dreidimensionaler, mit Nullen gefüllter Array *data_inputs* erstellt, wobei die ersten zwei Dimensionen die Breite und Höhe des Bildes darstellen und die dritte Dimension als Farbkanal dient. Danach werden für jeden vorhandenen Pinselstrich die im Zeitabstand Δt gespeicherten Positionen ermittelt und der Wert des *data_inputs* Arrays wird an dieser Stelle mit einer Eins ersetzt.

Das eben beschriebene Verfahren wird mit Code 3 veranschaulicht. Abbildung 2 zeigt ein Beispielbild eines Apfels, nach der Anwendung von diesem Verfahren auf die Rohdaten. Da die gezeichneten Punkte nicht kontinuierlich, sondern in einem gewissen Zeitabstand Δt von Google abgespeichert wurden, ist das Ergebnis noch zu wenig detailliert. Man erkennt zwar eine grobe Form, doch für meine Arbeit wird ein genaueres Bild benötigt. Der dabei verwendete Bresenham-Algorithmus und seine Implementierung in Python werden im folgenden Kapitel erläutert.

```
def Funktion(data_raw):
   data_inputs = torch.zeros(1, 256, 256) # erstellt einen 3 dimensionalen Array
    der Form: (Farbkanal, Höhe, Breite)

   for stroke in data_raw["drawing"]:
      for j in range(len(stroke[0])):
         xp = stroke[0][j]
         yp = stroke[1][j]
         data_inputs[0][yp][xp] = 1
         del xp
         del yp

   return data_inputs
```

Code 3: Funktion zur Einbettung der Rohdaten in eine einheitlichen Rastergrafik (ohne Anwendung des Bresenham-Algorithmus)

2.3 Anwendung des Bresenham-Algorithmus

Da die gegebenen Punkte eines Bildes jeweils der Reihe nach und in einzelne Pinselstriche gegliedert sind, kann man die ursprüngliche Zeichnung sehr gut rekonstruieren. Das zu lösende Problem wird mit Abbildung 3 veranschaulicht und besteht darin, aus zwei gegebenen Punkten

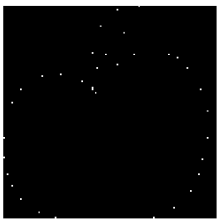


Abbildung 2: Exemplar eines Bildes vor Anwendung des Bresenham-Algorithmus

eine Linie zu berechnen, die mit einzelnen Pixeln dargestellt werden kann.

Einer der bekanntesten Algorithmen zur Lösung von diesem Problem ist der sogenannte Bresenham-Algorithmus [6], der mit einer relativ geringen Komplexität und mit einer sehr hohen Rechengeschwindigkeit punkten kann.

Code 4 berechnet im Gegensatz zu Code 3 auch eine Verbindungslinie der gegebenen Punkte. Diese Linien sind aneinanderhängend und werden nur durch einen neuen Pinselstrich unterbrochen. Dadurch kann aus sehr reduzierten Daten ein dem Original extrem nahekommendes Bild entstehen, wie Abbildung 4 veranschaulicht.

Meine Implementation des Bresenham-Algorithmus in Python beruht ausschliesslich auf einen Artikel der Webseite RogueBasin [2]. Die von mir verwendete Funktion *bresenham_line()* ist zudem im Anhang B ersichtlich.

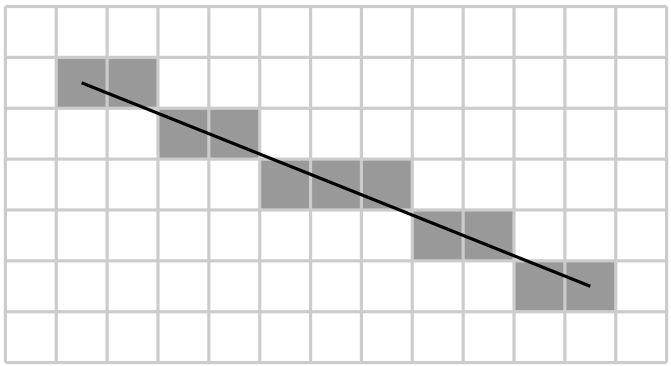


Abbildung 3: Rasterung einer Linie mit Anfangspunkt (1,1) und Endpunkt (11,5)

Quelle: `https://commons.wikimedia.org/wiki/File:Bresenham.svg`

Abbildung 4: Exemplar eines Bildes nach Anwendung des Bresenham-Algorithmus

```
def Funktion(data_raw):
   data_inputs = torch.zeros(1, 256, 256) # erstellt einen 3 dimensionalen Array
    der Form: (Farbenkanal, Höhe, Breite)

   for stroke in data_raw[i]["drawing"]:
       x0 = stroke[0][0]
       y0 = stroke[1][0]

       for j in range(0, len(stroke[0])):
           x1 = stroke[0][j]
           y1 = stroke[1][j]

           line = bresenham_line(x0, y0, x1, y1)   # die Funktion bresenham_line()
     benötigt zwei Anfangspunkt als Parameter und gibt eine Verbindunglinie aus

           for xp, yp in line:
               data_inputs[0][yp][xp] = 1
           x0 = x1
           y0 = y1

   return data_inputs
```

Code 4: Funktion zur Einbettung der Rohdaten in eine einheitlichen Rastergrafik (mit Anwendung des Bresenham-Algorithmus)

2.4 Bewertung des Datensatzes

Um eine gute künstliche Intelligenz zur Bildklassifizierung mit Hilfe des Gradientenabstiegsverfahrens (Kapitel 3.3) zu entwickeln, werden viele qualitativ hochwertige Trainingsdaten benötigt. Deshalb sollte man schon vor dem Trainieren eines neuronalen Netzes evaluieren, ob der gegebene Datensatz erfolgversprechend ist.

Der *Quick, Draw!* Datensatz bietet eine überwältigende Quantität. Durchschnittlich beinhaltet jede der insgesamt 345 Kategorien mehr als 144'000 verschiedene Trainingsbilder, was zu ungefähr 50 Million Zeichnungen von verschiedensten Gegenständen führt [1]. Im Vergleich

dazu, hat der auch äusserst bekannte MNIST Datensatz durchschnittlich nur ungefähr 7000 verschiedene Trainingsbilder pro Kategorie [14].

Zudem bietet der Datensatz von Google auch eine sehr gute Qualität. Nicht nur die Auflösung der Bilder von 256×256 Pixel, sondern auch das Speicherkapazität schonende Format tragen dazu bei. Dadurch ist es zwar leider etwas umständlich, die einzelnen Bilder vom .ndsjon Format in eine Rastergrafik umzuwandeln, jedoch konnte ich dieses Problem mit meinem Compiler (Anhang B) gut bewältigen.

3 Das neuronale Netz

In den Kapiteln 3.1, 3.2 und 3.3 wurde ein vereinfachtes neuronales Netz angenommen mit n Reizen in der Eingabeschicht und mit einer Aktivierung $\hat{y}(\vec{\alpha}, \beta)$ in der Ausgabeschicht, wobei sogenannte verdeckte Schichten (englisch hidden layers) vernachlässigt wurden. Abbildung 5 veranschaulicht diesen Sachverhalt.

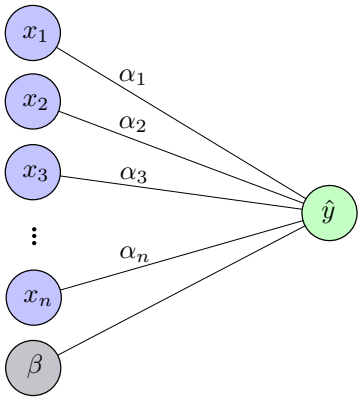


Abbildung 5: Vereinfachter Aufbau eines neuronalen Netzes

Dabei werden die Reize mit dem Vektor $\vec{X} = \begin{pmatrix} x_1 & x_2 & x_2 & \dots & x_n \end{pmatrix}^\top$, die Gewichte mit dem Vektor $\vec{\alpha} = \begin{pmatrix} \alpha_1 & \alpha_2 & \alpha_2 & \dots & \alpha_n \end{pmatrix}^\top$ und die gesamten Parameter des neuronalen Netzes mit dem Vektor $\vec{\theta} = \begin{pmatrix} \alpha_1 & \alpha_2 & \alpha_2 & \dots & \alpha_n & \beta \end{pmatrix}^\top$ dargestellt.

3.1 Die Aktivierung

Da es sich um ein lineares neuronales Netz handelt, lässt sich die Übertragung $\Psi(\vec{\alpha}, \beta)$ auf ein Neuron entweder als Summe oder mit dem Skalarprodukt wie folgt definieren:

$$\Psi(\vec{\alpha}, \beta) = \sum_{i=1}^{n} \alpha_i x_i + \beta = \langle \vec{\alpha}, \vec{X} \rangle + \beta \tag{1}$$

Für die Aktivierung $\hat{y}(\vec{\alpha}, \beta)$ des Ausgabeneurons wird noch eine Fehlerfunktion benötigt, wobei äusserst viele zur Auswahl stehen. Ich habe mich während meiner Arbeit für die im maschinellen Lernen weit verbreitete [5] und erfolgversprechende [9] Rectifier (auf Deutsch: Gleichrichter) Funktion entschieden, die auch oft als rectified linear unit (kurz ReLU) bezeichnet wird. Die ReLU Funktion ist wie folgt definiert:

$$R(x) = \max(0, x) = \begin{cases} x & \text{wenn } x > 0, \\ 0 & \text{wenn } x \leq 0. \end{cases} \tag{2}$$

Da die Ableitung der Aktivierungsfunktion später noch benötigt wird, möchte ich hier gleich

die Ableitung der ReLU Funktion definieren. Da die Funktion $R(x)$ nur für $R(x) = 0$ nicht differenzierbar ist, kann an dieser Stelle entweder die Ableitung $R'(x) = 0$ oder $R'(x) = 1$ willkürlich gewählt werden:

$$R'(x) = \begin{cases} 1 & \text{wenn } x > 0, \\ 0 & \text{wenn } x \leq 0. \end{cases} \tag{3}$$

Führt man nun zuerst die Übertragungsfunktion $\Psi(\vec{\alpha}, \beta)$ und danach die Aktivierungsfunktion $R(x)$ aus, so bekommt man die Aktivierung $\hat{y}(\vec{\alpha}, \beta)$ des Ausgabeneurons:

$$\hat{y}(\vec{\alpha}, \beta) = R(\Psi(\vec{\alpha}, \beta)) = \max(0, \langle \vec{\alpha}, \vec{X} \rangle + \beta) \tag{4}$$

Die Funktionen $\hat{y}(\vec{\alpha}, \beta)$ und $\Psi(\vec{\alpha}, \beta)$ wurden in Abhängigkeit zu $\vec{\alpha}$ und β geschrieben, da man später beim Gradientenabstiegsverfahren im Kapitel 3.3 nach den einzelnen Gewichten und nach dem Bias partiell ableiten muss.

3.2 Die Fehlerfunktion

Um ein neuronales Netz zu trainieren, muss die momentane Genauigkeit einer KI ermittelt werden können. Dafür wird die Fehlerfunktion $F(\vec{\alpha}, \beta)$ eingeführt:

$$F(\vec{\alpha}, \beta) = (\hat{y}(\vec{\alpha}, \beta) - y)^2 \tag{5}$$

Das Quadrieren der Differenz zwischen dem Ausgabewert $\hat{y}(\vec{\alpha}, \beta)$ und dem erwarteten Wert y ist äusserst wichtig für das folgende Gradientenabstiegsverfahren. Weil dadurch stets $F(\vec{\alpha}, \beta) \geq 0$ gilt, ist die Differenz und somit die Ungenauigkeit der KI beim globalen Minimum am geringsten. Deshalb versucht man beim Training des neuronalen Netzes dieses globale Minimum oder, was viel realistischer ist, eines der lokalen Minima der Fehlerfunktion $F(\vec{\alpha}, \beta)$ zu finden.

3.3 Das Gradientenabstiegsverfahren

Bei dem sogenannten Gradientenabstiegsverfahren versucht man sich durch wiederholtes Anwenden des Verfahrens über mehrere Epochen einem Minimum einer Funktion $F : \mathbb{R}^{n+1} \to \mathbb{R}$, $\vec{\alpha}, \beta \mapsto F(\vec{\alpha}, \beta)$ zu nähern. Da der Gradient $\nabla F(\vec{\alpha}, \beta)$ stets in die steilste Richtung an der momentanen Stelle zeigt, kann man sich schnellst möglich einem Minimum nähern, indem man diesen Gradienten vom Parametervektor $\vec{\theta}$ subtrahiert.

In den folgenden Unterkapiteln wird dieses Optimierungsverfahren zuerst im weniger dimensionalen Raum veranschaulicht und danach für das $(n+1)$-dimensinale neuronale Netz berechnet.

3.3.1 Veranschaulichung im weniger dimensionalen Raum

Zur Veranschaulichung lässt sich das Optimierungsproblem auf eine Funktion $g : \mathbb{R} \to \mathbb{R}$, $x \mapsto g(x)$ reduzieren, wobei im folgenden Beispiel die Polynomfunktion

$$g(x) = \frac{(x+3.5)(x+2)(x+0.5)(x-1)(x-3)(x+1)}{60} + 4 \tag{6}$$

gewählt wurde. Abbildung 6 zeigt, dass die Funktion $g(x)$ ein globales Minimum und zwei lokale Minima aufweist.

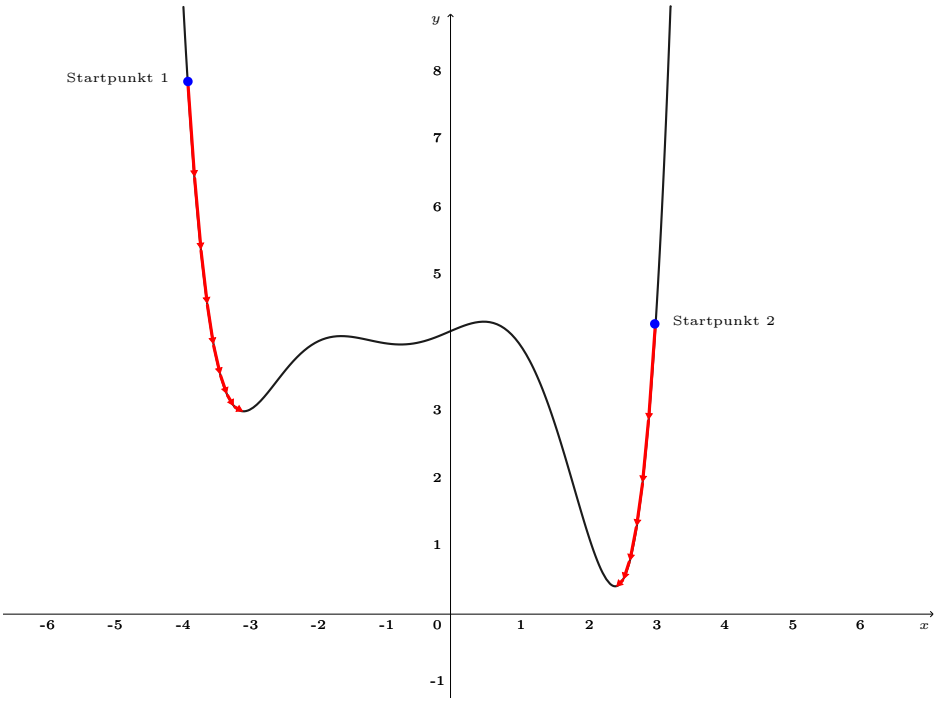


Abbildung 6: Das Gradientenabstiegsverfahren ohne Impuls bei der Polynomfunktion $g(x) = \frac{(x+3.5)(x+2)(x+0.5)(x-1)(x-3)(x+1)}{60} + 4$ mit zwei verschiedenen Startpunkten

Nun lässt sich durch das Gradientenabstiegsverfahren ein Minimum der Funktion finden, wobei ein beliebiger Startpunkt gewählt werden kann, was jedoch zu unterschiedlichen Resultaten führen könnte.

Wählt man einen zufälligen Startpunkt x_0 nähert man sich durch wiederholtes Subtrahieren der Ableitung vom letzten x Wert einem Minimum der Funktion $g(x)$, wenn man die Ableitung vor dem Subtrahieren mit einer passenden Lernrate λ multipliziert. Der x Wert zur Wiederholung $(t+1)$ ist rekursiv definiert als

$$x_{t+1} = x_t - \lambda g'(x_t), \tag{7}$$

wobei $t \geq 0$ und $\lambda > 0$.

Zudem sollte die Lernrate nicht zu gross sein, da dabei die Gefahr besteht, das Ziel zu überspringen. λ sollte jedoch auch nicht zu klein gewählt werden, da ansonsten der Algorithmus zu lange brauchen würde, um zu einem Minimum zu gelangen.

Wendet man dieses Verfahren später im maschinellen Lernen an, wird noch ein Impuls hinzukommen, der es ermöglicht schlechte lokale Minima zu überspringen. Würde man in Abbildung 6 bei Startpunkt 1 mit einem passenden Impuls das Verfahren anwenden, so würden die schlechten lokalen Minima übersprungen werden und man würde das globale Minimum erreichen.

Hängt nun eine Funktion $h : \mathbb{R}^2 \to \mathbb{R}$ von zwei Variablen ab, als Beispiel $h(x, y) = x^2 + y^2$, so lässt sich diese Funktion durch wiederholtes Subtrahieren des Gradienten $\nabla h(x, y)$ auch minimieren. Die Werte der Variablen zur Wiederholung $(t + 1)$ sind dann rekursiv definiert als

$$\begin{pmatrix} x_{t+1} \\ y_{t+1} \end{pmatrix} = \begin{pmatrix} x_t \\ y_t \end{pmatrix} - \lambda \nabla h(x_t, y_t), \tag{8}$$

wobei

$$\nabla h(x, y) = \begin{pmatrix} \frac{\partial h}{\partial x} \\ \frac{\partial h}{\partial y} \end{pmatrix}, \tag{9}$$

$t \geq 0$ und $\lambda > 0$.

Dieses Prinzip lässt nun sich auf ein neuronales Netz übertragen, um die Fehlerfunktion $F(\vec{\alpha}, \beta)$ zu minimieren, die von $(n + 1)$ verschiedenen Variablen abhängt.

3.3.2 Die Berechnung des Gradienten der Fehlerfunktion

Um beim Training eines neuronalen Netzes die Gewichte mit dem Gradientenabstiegsverfahren zu optimieren, muss der Gradient der Fehlerfunktion berechnet werden. Leitet man die Funktion $F(\vec{\alpha}, \beta)$ nach dem Gewicht α_n ab, so folgt aus Gleichung (3) und der Kettenregel:

$$\frac{\partial F(\vec{\alpha}, \beta)}{\partial \alpha_n} = \begin{cases} 2(\hat{y}(\vec{\alpha}, \beta) - y)x_n = 2(\langle \vec{\alpha}, \vec{X} \rangle + \beta - y)x_n & \text{wenn } \Psi(\vec{\alpha}, \beta) > 0, \\ 0 & \text{wenn } \Psi(\vec{\alpha}, \beta) \leq 0. \end{cases} \tag{10}$$

Analog ergibt sich die partielle Ableitung nach dem Bias β:

$$\frac{\partial F(\vec{\alpha}, \beta)}{\partial \beta} = \begin{cases} 2(\hat{y}(\vec{\alpha}, \beta) - y) = 2(\langle \vec{\alpha}, \vec{X} \rangle + \beta - y) & \text{wenn } \Psi(\vec{\alpha}, \beta) > 0, \\ 0 & \text{wenn } \Psi(\vec{\alpha}, \beta) \leq 0. \end{cases} \tag{11}$$

Der Gradient der Fehlerfunktion ist demnach definiert als

$$\nabla F(\vec{\alpha}, \beta) = \left(\frac{\partial F(\vec{\alpha}, \beta)}{\partial \alpha_1} \quad \frac{\partial F(\vec{\alpha}, \beta)}{\partial \alpha_2} \quad \frac{\partial F(\vec{\alpha}, \beta)}{\partial \alpha_3} \quad \dots \quad \frac{\partial F(\vec{\alpha}, \beta)}{\partial \alpha_n} \quad \frac{\partial F(\vec{\alpha}, \beta)}{\partial \beta} \right)^\top. \tag{12}$$

Ferner ist zu beachten, dass aufgrund der ReLU Funktion ein theoretisches Problem beim Optimierungsprozess besteht, wenn die Bedingung $\Psi(\vec{\alpha}, \beta) \leq 0$ zu früh durch Zufall anstatt durch wiederholtes Anwenden des Algorithmus erreicht wird. Dieses Problem ist auch bekannt

unter dem Namen *dying ReLU* [15] und es lässt sich in der Praxis auf verschiedenste Wege lösen. Obwohl ich während meiner praktischen Arbeit nie mit diesem Problem zu kämpfen hatte, möchte ich hier zumindest zwei Lösungsansätze besprechen.

Es gäbe zum Beispiel die sehr einfache Möglichkeit bei der zufälligen Initialisierung der Parameter für den Bias β eine genug grosse Zahl zu wählen, um zu vermeiden, dass der Fall $\Psi(\vec{\alpha}, \beta) \leq 0$ zu früh durch Zufall eintritt.

Zudem lässt sich das Problem aber auch mit einem sogenannten stochastischen Gradientenabstiegsverfahren (kurz SGD) beheben [15, Seite 6], was auch in PyTorch direkt in die Optimierungsfunktion integriert wurde. Dies könnte ein Grund dafür sein, dass ich bis zum theoretischen Teil meiner Arbeit gar nicht auf das *dying ReLU* Problem aufmerksam wurde, denn ich durfte von Anfang an die Vorzüge des SGD geniessen.

3.3.3 Der Impuls

Der Impuls μ gibt an, wie stark beim Training eines neuronalen Netzes die Gradienten der vorherigen Epochen berücksichtigt werden sollen. Ein passender Impuls erlaubt es dem Algorithmus beim Gradientenabstiegsverfahren schlechte lokale Minima viel einfacher zu überspringen und somit mit höherer Wahrscheinlichkeit dem globalen Minimum näher zu kommen.

Meine empirische Erfahrung zeigt, dass ein Impuls von ungefähr 0.9 gute Ergebnisse verspricht, wobei $0 \leq \mu \leq 1$. Diese Erkenntnis wird unter anderem auch von Ning Qian [18, Seite 149] bestätigt.

3.3.4 Berechnung der neuen Parameter

Mit dem Gradienten der Fehlerfunktion und der Einführung eines Impulses μ lassen sich nun die Parameter $\vec{\theta}$ einer beliebigen Epoche $(t+1)$ des Trainings rekursiv definieren als

$$\vec{v}_{t+1} = \mu\vec{v}_t - \lambda\nabla F(\vec{\alpha}_t, \beta_t) \tag{13a}$$

$$\vec{\theta}_{t+1} = \vec{\theta}_t + \vec{v}_{t+1}, \tag{13b}$$

wobei

$$\vec{v}_0 = \vec{0},$$

$$t \geq 0,$$

$$\lambda > 0,$$

$$\mu \in [0, 1].$$

In der Programmbibliothek PyTorch [3] basiert die Implementation des Gradientenabstiegsverfahrens auf den Formeln aus der Arbeit *On the importance of initialization and momentum*

in deep learning. Dadurch entsprechen die von mir hergeleiteten Gleichungen 13 den von PyTorch und somit auch den für meine praktische Arbeit verwendeten Formeln [19, Seite 2] für den Optimierungsalgorithmus.

4 Optimierung des Trainings

Um eine erfolgversprechende künstliche Intelligenz zu entwickeln, braucht es mehr als einen grundsätzlich funktionierenden Lernalgorithmus, denn dieser beruht auf verschiedenen Parameter, die anfangs definiert werden müssen.

Beispiele für solche Parameter, die in den folgenden Unterkapiteln besprochen werden, sind die Auflösung der Trainingsbilder, die Lernrate und der Impuls. Die Suche nach den passenden sogenannten Hyperparametern wird auch Hyperparameteroptimierung genannt.

Doch auch die Struktur des neuronalen Netzes spielt eine wesentliche Rolle, wobei eine passende Anzahl von verdeckten Schichten und von Neuronen in diesen Schichten zu finden ist. Zudem kann auch das Problem des sogenannten Overfittings auftreten, was sich jedoch ziemlich leicht mit Dropout Schichten beheben lässt.

4.1 Fluch der Dimensionalität

Um die optimale Auflösung der Trainingsdaten zu finden, muss man zuerst den Begriff *Fluch der Dimensionalität* verstehen. Damit wird beschrieben, wie sich die Anzahl der benötigten Beobachtungen N, um im p-dimnensionalen Raum sinnvolle Aussagen zu treffen beim Erhöhen beziehungsweise Verringern der Dimensionen verändert.

Man nehme beispielhaft eine zufällige Beobachtung einer natürlichen Zahl zwischen 0 und 9 an, so beträgt die Wahrscheinlichkeit, dass diese Zahl eine 7 ist statistisch gesehen $\frac{1}{10}$. Erweitert man dies um eine Dimension, so besteht eine zufällige Beobachtung nun aus zwei natürlichen Zahlen zwischen 0 und 9. Dadurch verringert sich die Wahrscheinlichkeit, dass diese Beobachtung auf den ausgewählten Punkt $(7, 0)$ fällt auf $\frac{1}{100}$. Um mit gleicher Wahrscheinlichkeit wie in einer Dimension auf dem Punkt $(7, 0)$ zu landen, müssen nun 10 verschiedene Beobachtungen gemacht werden, wodurch die gleiche Stichprobendichte erreicht wird.

Dieses Problem findet sich auch im maschinellen Lernen wieder und lässt sich mit der folgenden Formel [8, Seite 23] beschreiben:

$$N \propto \varrho^p, \tag{14}$$

wobei N die Anzahl Trainingsdaten, p die Anzahl Dimensionen des neuronalen Netzes und ϱ die Dichte der Trainingsdaten beschreibt.

Man nehme beispielsweise an, dass ein zweidimensionales neuronalen Netz (ein Input und ein Bias) $N_2 = 100$ Trainingsdaten für ein gewisses Ergebnis benötigt. Dann werden laut Gleichung (14) mit einem 20-dimensionalem neuronalen Netz (19 Inputs und ein Bias) für dasselbe Ergebnis $N_{20} = N_2^{\frac{p_{20}}{p_2}} = 100^{10}$ Trainingsdaten benötigt.

Deshalb sollte man grundsätzlich versuchen die Dimensionen des zu trainierenden neuronalen Netzes möglichst gering zu halten, damit eine genug hohe Dichte der Trainingsdaten besteht. Jedoch darf man nicht vergessen, dass eine gewisse Dimensionalität benötigt wird, um die Komplexität des Problems der Bildklassifizierung zu bewältigen.

4.2 Grösse und Auflösung der Trainingsdaten

Weil das neuronale Netz für jeden Pixel eines Bildes ein einzelnes Eingabeneuron besitzt, verändert sich die Dimensionalität der Fehlerfunktion je nach Auflösung der Bilder. Da die Zeichnungen nicht nur beim Training, sondern auch bei der Evaluation der KI erst vor der Eingabe ins neuronale Netz skaliert werden, können stets 256×256 grosse Bilder von der KI klassifiziert werden, unabhängig von der für das Training ausgewählten Auflösung.

Skaliert man die Trainingsdaten gar nicht, so hat man zum Beispiel bei der Klassifizierung von 10 verschiedenen Gegenständen mindestens $256^2 * 10 = 655'360$ Gewichte, die trainiert werden müssen. Skaliert man diese Bilder auf eine Auflösung von 16×16, so verringert sich die Dimensionalität auf mindestens $16^2 * 10 = 2'560$, wodurch sich die Dichte des Datensatzes aufgrund des Fluchs der Dimensionalität erhöht und die Trainingszeit sich verringert.

Doch die empirische Erfahrung hat gezeigt, dass bei einer Skalierung auf 16×16 Pixel zu viele Informationen verloren gehen, wodurch die KI bei der Klassifizierung von Testdaten keine zufriedenstellenden Ergebnisse liefern konnte. Deshalb habe ich weitere Versuche mit den Auflösungen 64×64 und 32×32 Pixel durchgeführt, wobei die letztere bei den Testdaten die besten Ergebnisse versprach.

Die Auflösung der Bilder eines Datensatzes lässt sich im Compiler (Anhang B) mit der Tupel *img_dim* festlegen und kann je nach Bedarf auch in eine Richtung gestreckt werden.

4.3 Die Struktur des neuronalen Netzes

Die Anzahl der verdeckten Schichten und der sich darin befindenden Neuronen hat auch einen massgeblichen Einfluss auf die Trainingszeit eines neuronalen Netzes und auf die Genauigkeit der KI. Nicht nur im Rahmen meiner Arbeit, sondern auch in der gängigen Praxis werden diese Zahlen oft durch Versuch und Irrtum gefunden, denn während zu viele verdeckte Schichten und Neuronen die Trainingszeit ins Unermessliche steigen lassen, führen zu wenige zu unzureichender Genauigkeit bei Testdaten [17].

Da ein flexibler Umgang mit diesen Zahlen benötigt wird, um empirische Erfahrungen zu sammeln, habe ich in dem Trainingsprogramm zwei zusätzliche Variablen eingeführt. Die Variable $hidden\_layers\_num \in \mathbb{N}_0$ gibt an, wie viele verdeckte Schichten vorhanden sind und die Variable $layers\_connection\_factor > 0$ gibt an, um welchen Faktor die Anzahl Neuronen von Schicht zu Schicht abnimmt. Bei Bedarf kann die Anzahl Neuronen auch zunehmen, was teilweise auch überraschende Ergebnisse liefern kann. Zudem ist die Ausgabeschicht nicht davon betroffen, denn dort befindet sich stets ein Neuron pro Gegenstand.

Im Rahmen meiner Arbeit konnte ich bei 33 verschiedenen Gegenständen mit 2 verdeckten Schichten und einem Verbindungsfaktor von 0.5 bei noch nie zuvor gesehenen Validierungsdaten eine Genauigkeit von 84% erreichen. Mit derselben Struktur und 10 verschiedenen Gegenständen konnte die KI eine Genauigkeit von 91% erreichen. Bei nur 4 Gegenständen genügte eine verdeckte Schicht mit Verbindungsfaktor 0.5 um eine Genauigkeit von 98% zu erreichen. Eine genauere

Besprechung der Ergebnisse folgt im Kapitel 5.

4.4 Die Lernrate und der Impuls

Beim Gradientenabstiegsverfahren sind die Lernrate λ und der Impuls μ die zwei wichtigsten Hyperparameter und haben somit auch einen grossen Einfluss auf den Erfolg des Trainings einer KI. Obwohl die Suche nach diesen Parametern im modernen Gebrauch schon oft automatisiert wird [7], ist im Rahmen meiner Arbeit auch hier wieder die heuristische Methode der beste Ansatz.

Wie schon im Unterkapitel 3.3.3 genauer erläutert, habe ich mich nach einigem Ausprobieren und Recherchieren meistens für einen Impuls von $\mu = 0.9$ entschieden.

Im Vergleich dazu ist die Lernrate ein viel sensiblerer Hyperparameter, der meiner Erfahrung nach stark von der Batch Grösse und von der Struktur des neuronalen Netzes abhängt. Während eine zu gross gewählte Lernrate dazu führen kann, dass gute Minima der Fehlerfunktion übersprungen werden und man nicht an ein sinnvolles Ziel gelangt, treibt eine zu kleine Lernrate die Trainingszeit immer weiter in die Höhe und erhöht das Risiko, in einem schlechten lokalen Minima zu enden. Deshalb konnte ich mich auch nach vielen Versuchen mit Lernraten von 0.001 bis 0.5 nicht auf ein λ festlegen, das stets gute Resultate versprechen konnte.

Mir ist jedoch aufgefallen, dass mit steigender Anzahl Epochen (unter der Annahme, dass der Wert der Fehlerfunktion sinkt) der Wechsel auf eine kleinere Lernrate einen positiven Effekt auf den Fehlerwert und somit auch einen deutlichen Sprung der Genauigkeit zur Folge haben kann. Eine solche Entwicklung ist in Abbildung 7 ersichtlich.

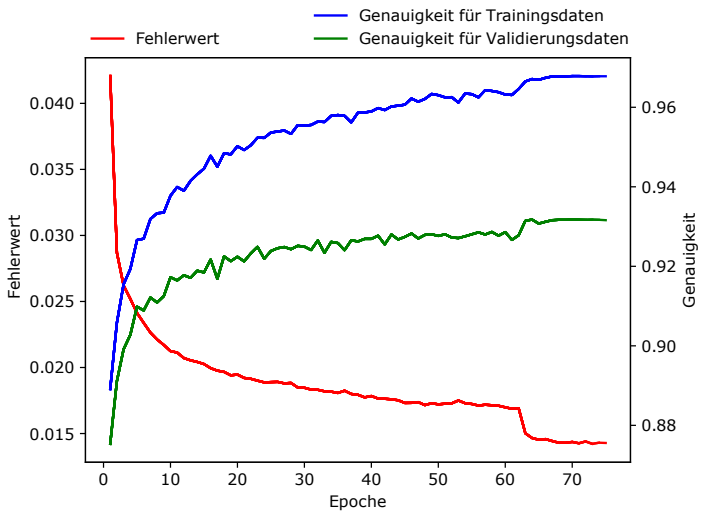


Abbildung 7: Entwicklung eines neuronalen Netzes mit einem manuellen Wechsel der Lernrate während des Trainings (Diagramm entspricht dem Resultat mit ID 16 in Abbildung 9)

Deshalb habe ich die Möglichkeit einer dynamischen Lernrate eingeführt, die mit drei Variablen gesteuert werden kann, wobei die torch.optim.lr_scheduler.StepLR() Funktion aus PyTorch benutzt wurde [3]. Die boolesche Variable *lr_scheduler* gibt an, ob die Lernrate mit der Zeit abnehmen soll und sofern dies der Fall ist, gibt die Variable *lr_stepsize* an, nach wie vielen Epochen die Lernrate um den Faktor *gamma* reduziert werden soll. Die genaue Implementation in meinem Programm ist im Anhang C ersichtlich.

4.5 Dropout Schichten

Eine weitere Gefahr, die während des Trainings eines neuronalen Netzes auftreten kann, ist das Problem des Overfittings. Es beschreibt die Entwicklung einer stets steigenden Genauigkeiten für die Trainingsdaten und anfangs auch für die Validierungsdaten, doch letztere nimmt ab einem gewissen Punkt wieder abnimmt, da sich das neuronale Netz zu stark auf den vorgegebenen Datensatz spezialisiert. Dieses Problem macht sich zwar vor allem bei eher kleinen Datensätzen bemerkbar [11] (was man vom *Quick, Draw!* Datensatz nicht behaupten kann), doch trotzdem gab es einzelne Fälle des Overfittings während meiner praktischen Arbeit, wie Abbildung 8 zeigt, weshalb ich mich dazu entschied auch dafür eine Lösung einzubauen.

Das Einfügen von sogenannten Dropout Schichten, die zufällige Elemente des Input Arrays vor der Eingabe in die folgende lineare Schicht auf Null setzen, beheben ziemlich effektiv das Problem des Overfittings [11]. In PyTorch lassen sich solche Schichten mit der Klasse *torch.nn.Dropout()* einfügen [3] und werden im meinem Programm mit der booleschen Variable *dropout* aktiviert. Sofern der Dropout aktiviert ist, gibt die Variable *dropout_rate* die Wahrscheinlichkeit eines Elementes an, auf Null gesetzt zu werden. Die genaue Implementation ist wiederum im Anhang C ersichtlich.

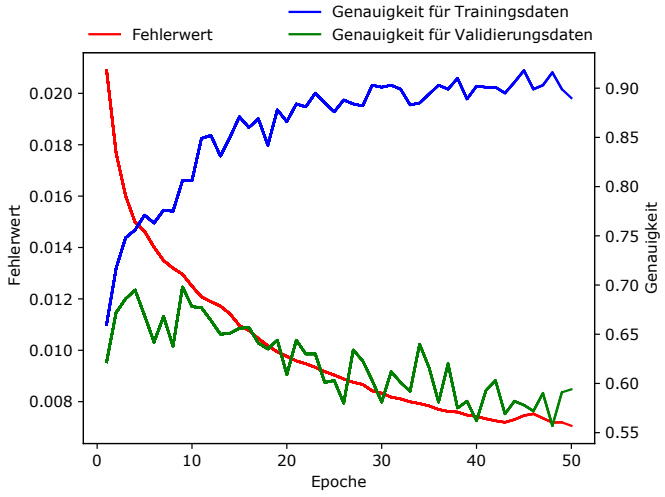


Abbildung 8: Overfitting bei einem neuronalen Netz (Diagramm entspricht dem Resultat mit ID 3 in Abbildung 9)

5 Ergebnisse

Abbildung 9 zeigt eine Tabelle mit den interessantesten Ergebnissen meiner praktischen Arbeit. Wie schon im Kapitel 4 erläutert, muss beim Trainieren einer KI viel ausprobiert werden, bis man erste brauchbare Ergebnisse erhält. Deshalb wurden Resultate ohne grosse Aussagekraft aus Abbildung 9 entfernt. Es ist jedoch zu beachten, dass damit nicht alle negativen Resultate gemeint sind, da solche oft auch eine gewisse Aussagekraft beinhalten und somit teilweise auch in der Tabelle vertreten sind.

Jedes trainierte neuronale Netz ist mit einer einzigartigen ID versehen. Die Ergebnisse sind nach Anzahl Gegenstände (2. Spalte) und nach Genauigkeit bei den Validierungsdaten sortiert. Die Spalten zu den Genauigkeiten bei Trainings- und Validierungsdaten sind mit GktT und GktV abgekürzt. Man erkennt deutlich, wie die Genauigkeit der KI mit steigender Anzahl der zu klassifizierenden Gegenstände sinkt. Zudem erhöht sich die Genauigkeit tendenziell mit einer längerem Trainingsdauer.

Die Auflösung beschreibt bloss, wie die Bilder vor der Eingabe ins neuronale Netz skaliert wurden. Dies hat keinen Einfluss beim Evaluieren von komplett neuen Daten, die stets eine Auflösung von 256×256 Pixel haben sollten.

Die Trainingsdaten bilden 80% des Datensatzes. Folglich tragen Test- und Validierungsdaten jeweils 10% zu den Anzahl Daten bei.

Während einer Epoche wird das Gradientenabstiegsverfahren für jeden Batch genau ein Mal durchgeführt. Die Trainingsdaten werden deshalb in einzelne Pakete aufgeteilt, wobei die Batch Grösse auch in der Tabelle angegeben ist.

Die Struktur zeigt an, wie viele verdeckte Schichten das verwendete neuronale Netz besitzt und wie viele Neuronen sich jeweils darin befinden. Sie folgt dem Format [Inputneuronen, Neuronen verdeckte Schicht 1, Neuronen verdeckte Schicht 2, ..., Neuronen verdeckte Schicht x, Outputneuronen]. Folglich lässt sich daraus, sofern mindestens eine verdeckte Schicht vorhanden ist, auch der *layers_connection_factor* berechnen.

Sind bei der Lernrate zwei verschiedene Werte angegeben, so wurde diese während des Trainings manuell gewechselt. In der Spalte Lr Decay werden, sofern der lr_scheduler aktiviert ist, die genauen Werte zu den Variablen *lr_stepsize* und *gamma* angegeben.

ID	Anzhl Geg.	GktT in [%]	GktV in [%]	Auflösung	Anzahl Daten	Epochen	Laufzeit	Struktur	Batch Grösse	Lernrate	Lr Decay (stepsize/ gamma)	Impuls	Dropout
0	4	97.1	96.4	32 x 32	100'000	5	27min	[1024, 512, 4]	1	0.001	False	0.9	False
11	4	99.3	97.4	32 x 32	100'000	31	1h 10min	[1024, 1228, 1474, 1769, 2123, 2548, 3057, 3669, 4403, 4]	500	0.5	False	0.9	False
4	4	99.6	97.5	32 x 32	100'000	9	2h 24min	[1024, 512, 4]	1	0.01	False	0.9	False
28	4	98.6	98	32 x 32	100'000	55	1h 5min	[1024, 1024, 1024, 1024, 1024, 1024, 4]	1000	0.9	10/0.9	0.9	0.6
27	4	99.1	98.1	32 x 32	100'000	50	1h 2min	[1024, 1024, 1024, 1024, 1024, 1024, 4]	500	0.5	1/0.9	0.9	0.4
7	10	94.8	90.5	32 x 32	100'000	119	1h 48min	[1024, 307, 92, 27, 10]	100	0.01	False	0.9	False
19	10	93.8	91.1	32 x 32	100'000	82	1h 47min	[1024, 512, 256, 128, 64, 10]	100	0.5	1/0.95	0.9	0.5
5	10	100	91.6	32 x 32	100'000	105	5h 29min	[1024, 512, 256, 10]	1	0.01	False	0.9	False
22	10	94.3	91.8	32 x 32	200'000	77	2h 24min	[1024, 819, 655, 524, 419, 10]	1000	0.9	1/0.9	0.9	0.3
9	10	99.4	91.9	32 x 32	100'000	23	1h 9min	[1024, 2048, 4096, 8192, 16384, 10]	200	0.5	False	0.9	False
32	10	95.2	92.5	32 x 32	200'000	115	3h 34min	[1024, 512, 256, 128, 10]	300	0.5	3/0.9	0.9	0.5
8	10	98.7	92.5	32 x 32	100'000	12	37min	[1024, 2560, 6400, 16000, 10]	100	0.5	False	0.9	False
15	10	96.4	92.8	32 x 32	200'000	62	2h 3min	[1024, 512, 256, 128, 10]	10	0.1	False	0.9	0.4
21	10	95.6	92.8	32 x 32	200'000	57	3h 33min	[1024, 512, 256, 128, 64, 10]	5	0.1	5/0.5	0.9	0.3
14	10	95.8	92.8	32 x 32	200'000	35	1h 7min	[1024, 512, 256, 128, 10]	10	0.05	False	0.9	0.4
33	10	95.7	92.9	32 x 32	200'000	33	1h 26min	[1024, 1024, 1024, 1024, 1024, 1024, 10]	100	0.5	5/0.5	0.9	0.4
20	10	96.2	93	32 x 32	200'000	102	4h 6min	[1024, 512, 256, 128, 64, 10]	10	0.1	3/0.9	0.9	0.4
25	10	96	93.2	32 x 32	200'000	87	3h 37min	[1024, 1024, 1024, 1024, 1024, 1024, 10]	10	0.1	5/0.5	0.9	0.4
16	10	96.8	93.2	32 x 32	200'000	67	2h 24min	[1024, 512, 256, 128, 10]	10	0.1/0.01	False	0.9	0.4
31	10	96.1	93.3	32 x 32	200'000	62	6h 11min	[1024, 1024, 1024, 1024, 1024, 1024, 10]	10	0.1	5/0.5	0.9	0.4
13	12	98.6	90.4	32 x 32	100'000	177	2h 54min	[1024, 512, 256, 12]	100	0.1	False	0.9	0.3
2	19	100	85.7	32 x 32	10'000	24	53min	[1024, 512, 19]	1	0.1	False	0.9	False
30	20	88.3	86.4	32 x 32	500'000	27	43h 47min	[1024, 1024, 1024, 1024, 1024, 20]	1	0.01	3/0.9	0.9	0.5
29	20	89.8	87.2	32 x 32	500'000	15	9h 55min	[1024, 1228, 1474, 1769, 20]	1	0.01	2/0.9	0.9	0.4
3	33	89.7	57.5	32 x 32	100'000	50	3h 57min	[1024, 512, 33]	1	0.1	False	0.9	False
10	33	66.1	65.9	32 x 32	1'000'000	7	1h 18min	[1024, 1024, 1024, 1024, 33]	100	0.05	False	0.9	False
1	33	96.6	75	32 x 32	100'000	53	2h 17min	[1024, 512, 256, 33]	1	0.1	False	0.9	False
17	33	82.1	80.9	32 x 32	1'000'000	14	2h 21min	[1024, 716, 501, 351, 33]	10	0.1	False	0.95	0.4
23	33	83.6	82.2	32 x 32	1'000'000	58	10h 6min	[1024, 716, 501, 351, 33]	50	0.5	1/0.9	0.9	0.4
18	33	84.2	82.6	32 x 32	1'000'000	34	5h 35min	[1024, 716, 501, 351, 33]	10	0.05	False	0.95	0.4
6	33	90.6	84	32 x 32	1'000'000	21	13h 16min	[1024, 512, 256, 33]	1	0.01	False	0.9	False
24	33	86.3	84.5	32 x 32	1'000'000	43	14h 43min	[1024, 1228, 1474, 1769, 33]	10	0.1	1/0.9	0.9	0.4
12	33	94.6	85.5	32 x 32	1'000'000	72	11h 44min	[1024, 1024, 1024, 1024, 33]	100	0.1	False	0.9	False
26	33	88.3	85.7	32 x 32	1'000'000	10	31h 14min	[1024, 1024, 1024, 1024, 1024, 1024, 33]	1	0.05	1/0.7	0.9	0.2

Abbildung 9: Tabelle der Ergebnisse

5.1 Konfusionsmatrix

Eine Konfusionsmatrix bietet die Möglichkeit genau zu bestimmen, mit welcher Wahrscheinlichkeit einzelne Gegenstände von der KI richtig klassifiziert werden. Zudem lässt sich auch leicht herauslesen mit welcher Wahrscheinlichkeit ein Gegenstand A mit einem Gegenstand B verwechselt wird. Folglich lassen sich dank der Kunfusionsmatrix die Stärken und Schwächen einer trainierten KI genaustens bestimmen.

Die Zeilen einer Konfusionsmatrix geben den wahren Gegenstand an, während die Spalten für die Klassifizierung der KI stehen. Ein perfektes neuronales Netz hätte in der Konfusionsmatrix bis auf eine mit Einsen gefüllte Diagonale nur Nullen.

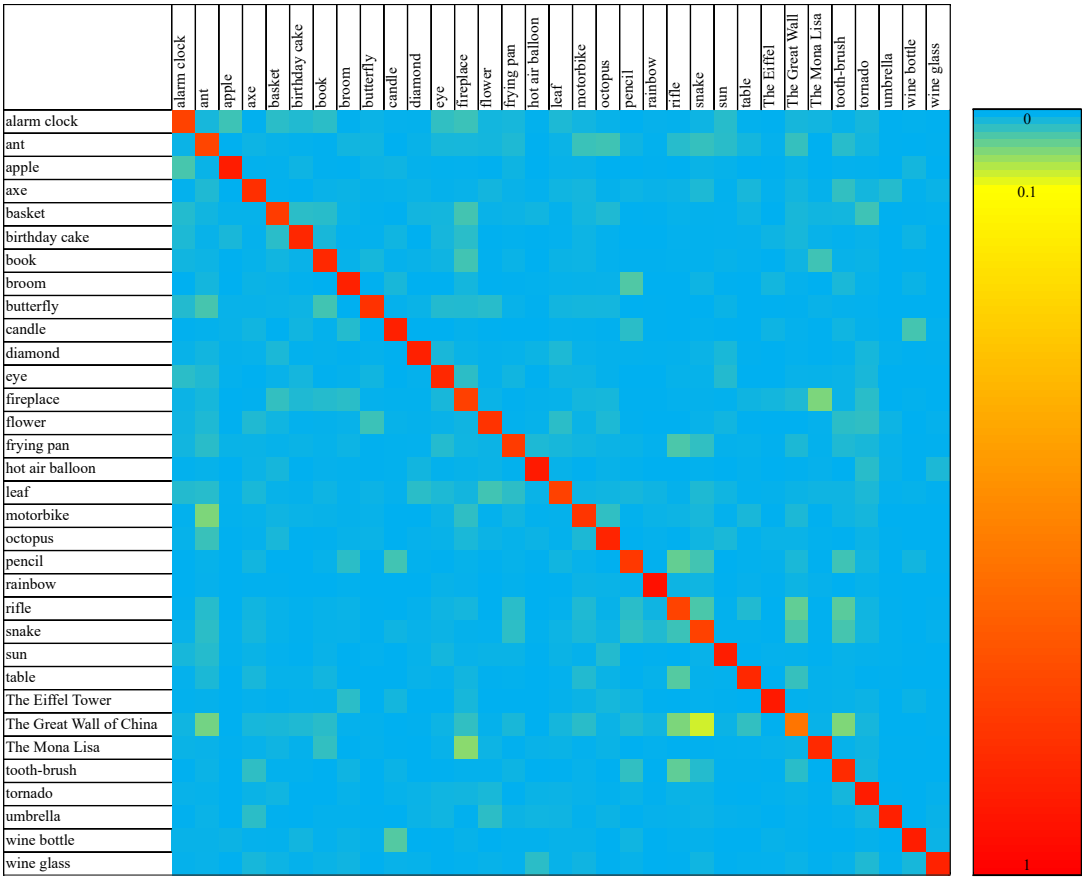


Abbildung 10: Visualisierung einer Konfusionsmatrix für 33 verschiedene Gegenstände aus dem Validierungsdatensatz (Berechnungen entsprechen dem neuronalen Netz mit ID 6 in Abbildung 9)

Eine grosse Konfusionsmatrix kann zwar schnell unübersichtlich werden, doch die Darstellung als Heatmap bietet eine gute Lösung. Abbildung 10 zeigt die Konfusionsmatrix eines neurona-

len Netzes, das zur Klassifikation von 33 verschiedenen Gegenständen trainiert und auf dem Validierungsdatensatz ausgewertet wurde.

Es ist anzumerken, dass der Farbverlauf nicht linear ist. Er startet mit der Farbe Blau beim Wert 0 und verläuft dann über Gelb beim Wert 0.1 zu Rot beim Wert 1. Folglich erscheinen Werte zwischen 0 und 0.1 grünlich, während Werte zwischen 0.1 und 1 eher orangefarben abgebildet sind.

Aus Abbildung 10 lassen sich nun sehr viele interessante Erkenntnisse gewinnen. Man vergleiche zum Beispiel die Zeile zum Eiffelturm (The Eiffel Tower) A mit der Zeile zur Chinesischen Mauer (The Great Wall of China) B. Es fällt sofort auf, dass die Zeile A bis auf das diagonale Feld fast durchgehend blau ist, während die Zeile B von einigen hellgrünen und sogar von einem gelben Feld unterbrochen wird. Zudem ist in Zeile B das Feld der Diagonalen besonders orangefarben.

Folgt man dem gelben Feld aus Zeile B, so wird ersichtlich, dass eine Chinesische Mauer von der KI auffällig oft fälschlicherweise als Schlange (snake) klassifiziert wurde. Doch spätestens wenn man Bilder aus dem Datensatz dieser beiden Kategorien miteinander vergleicht (Abbildung 11), wird ersichtlich, weshalb es der KI so schwer fällt, die Chinesische Mauer korrekt zu klassifizieren.

Die genauen Zahlen hinter der Konfusionsmatrix aus Abbildung 10 finden sich im Angang A.

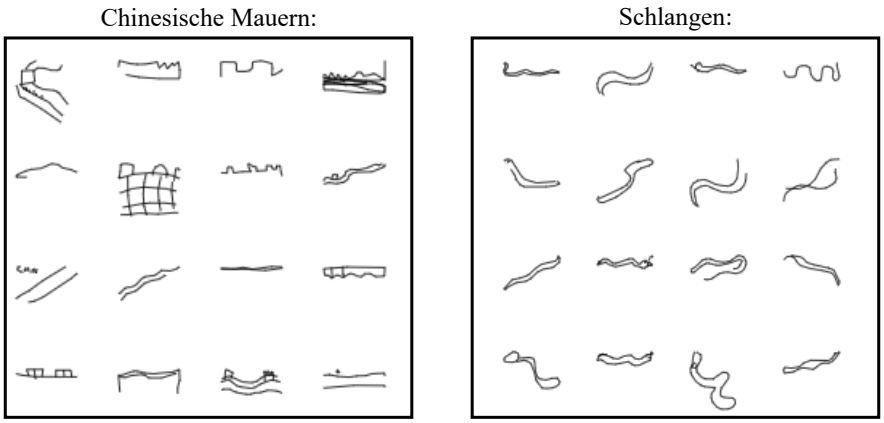


Abbildung 11: Validierungsdaten von Chinesischen Mauern und Schlangen im Vergleich

5.2 Leistung in der realen Welt

Da der verwendete Datensatz aus Skizzen von verschiedensten Menschen aus aller Welt besteht (wobei jeder dazu beitragen kann) [13] und nicht nur von einer ausgewählten Gruppe erstellt wurde, gibt es grundsätzlich wenige Begrenzungen der KI in der realen Welt.

Man nehme an, dass Daten aus der realen Welt stets auf einer 256×256 Pixel grossen Leinwand mit einem Pinsel, der einen Durchmesser von einem Pixel besitzt gezeichnet werden. So besteht der einzige Unterschied zwischen Validierungsdaten und Daten aus der realen Welt in der Grösse

der Zeichnungen. Denn Google hat die Bilder im Datensatz einheitlich so weit vergrössert, bis entweder horizontal oder vertikal ein Maximum erreicht wurde, wobei die Auflösung von 256×256 Pixel stets beibehalten wurde. Dies ist unter anderem auch in Abbildung 4 ersichtlich.

Solange man während dem erstellen von realen Daten auf die eben beschriebenen Annahmen achtet und möglichst gross skizziert, sollten sich die Ergebnisse der KI bei diesen Daten kaum von den Ergebnissen bei Validierungsdaten unterscheiden. Zum erstellen von neuen, realen Daten empfehle ich das kostenlose Programm paint.net.

Um selbst erstellte Bilder mit einem trainierten neuronalen Netz auszuwerten, wird das Programm zum Quellcode im Anhang E verwendet. Die ID des dabei verwendeten neuronalen Netzes wird mit der Variable *nn_model_number* angegeben. Ist die booleschen Variable *eval_my_data* auf True gesetzt, so wird das Bild myDrawing.png abgerufen und evaluiert. Ansonsten werden beim Ausführen des Programms zufällige Bilder aus dem Validierungsdatensatz ausgewertet.

6 Fazit

Das Hauptziel meiner praktischen Arbeit war das Erstellen eines Programms zur Klassifizierung von handgezeichneten Gegenständen mit Hilfe von künstlichen neuronalen Netzen. Da sich dieses Ziel in verschiedene Aufgaben aufteilen lässt, ergaben sich schlussendlich vier unterschiedliche Programme. Das erste übernimmt die Aufgabe des Zusammenstellens eines Datensatzes, das zweite trainiert dann ein neuronales Netz auf einen ausgewählten Datensatz und mit dem dritten und vierten Programm, lassen sich Daten mit bereits trainierten Netzen evaluieren. Da Google mit dem *Quick, Draw!* Datensatz eine enorme Datenmenge bereitstellt und meine Programme bezüglich der Quantität der Trainingsdaten und der Struktur des neuronalen Netzes sehr flexibel gestaltet wurden, sehe ich noch viel Potenzial in diesem Projekt, was sich im Rahmen einer Maturaarbeit leider nicht vollkommen ausschöpfen lässt. Und trotzdem bin ich mit den bereits erreichten Ergebnissen äusserst zufrieden. Die Spitzengenauigkeit von 98.1% wurde zwar bei nur vier verschiedenen Gegenständen erreicht, doch auch die weitaus schwierigere Aufgabe der Klassifizierung von 33 verschiedenen Gegenständen konnte noch mit einer Genauigkeit von über 85.7% bewältigt werden, wobei Stolpersteine wie die Chinesische Mauer miteinbezogen wurden. Ist man sich dabei noch bewusst, dass sich die Validierungsdaten kaum von komplett neuen Daten unterscheiden werden, finde ich die Ergebnisse umso bemerkenswerter. Als Vergleich erreichten Melanie Andersson u. a. mit künstlichen neuronalen Netzen bei sieben verschiedenen Gegenständen aus dem *Quick, Draw!* Datensatz eine Genauigkeit von 94.2% [4]. Zudem gelang es Kristine Guo u. a. bei der Klassifizierung von allen 345 Kategorien eine Genauigkeit von 62% zu erreichen [10]. Dabei wurden in beiden erwähnten Fällen unter anderem auch sogenannte Convolutional Layers eingebaut, während in meiner Arbeit nur klassische lineare Schichten verwendet wurden.

Das Ziel meiner schriftlichen Arbeit war es, sowohl einen Einblick in die praktische Arbeit zu liefern, als auch einen Teil der Mathematik des maschinellen Lernens zu erforschen. Da ein wesentlicher Teil meiner praktischen Arbeit darin bestand, den vorhandenen Datensatz neu zu strukturieren und dies meist grundsätzlich zum maschinellen Lernen dazu gehört, habe ich mich dazu entschieden, auch dies in meine schriftliche Arbeit einfliessen zu lassen. Dementsprechend wurde in Kapitel 2 zuerst der Umgang mit dem riesigen Datensatz von Google besprochen. Danach wurde im folgenden Kapitel 3 ein mathematisches Verständnis für künstliche neuronale Netze und deren Lernprozess aufgebaut, welches sich mit der Implementation in PyTorch deckt. Dies diente vor allem zum besseren Verständnis der Themen, die in den Kapiteln 4 und 5 folgten, wobei verschiedene, von mir genutzte, Optimierungsmöglichkeiten veranschaulicht und die damit erreichten Ergebnisse besprochen wurden.

7 Reflexion

Während des Erstellens der Programme und des Schreibens dieser Arbeit konnte ich extrem viel lernen. Als ich mit meiner Maturaarbeit anfing, war ich im Bezug auf das Programmieren noch ein Anfänger. Ich hatte zwar schon grobe Grundkenntnisse der Programmiersprache Python, doch dies war mein erstes grösseres Projekt, weshalb es doch auch einiges gibt, was ich heute anders machen würde. Meine vier Programme funktionieren nun zwar einwandfrei, doch besonders elegant sind die Quellcodes keinesfalls. Ich habe zum Beispiel gelernt, dass es vorteilhafter gewesen wäre, mit mehr verschiedenen Funktionen und mit mehr lokalen, anstatt der vielen globalen Variablen, zu arbeiten. Dahingegen bin ich mit meinem Zeitmanagement umso zufriedener, denn obwohl ich nur einen groben Zeitplan hatte, da das Endziel anfangs noch offen war, konnte ich mir stets ziemlich sicher sein, dass ich in keinen zeitlichen Verzug geraten werde.

Im Bezug auf meine schriftliche Arbeit, bin ich erfreut, dass ich mich trotz der Verwendung von PyTorch dazu entschieden haben, die Grundlagen der Mathematik von künstlichen neuronalen Netze zu studieren. Das tiefere Verständnis des Themas erlaubte es mir, bei der Optimierung des Trainings deutlich effektivere Methoden zu verwenden und bei der Analyse der Ergebnisse interessantere Erkenntnisse zu gewinnen. Zudem hatte ich die hervorragende Möglichkeit, das Arbeiten mit LaTeX kennenzulernen. Ich bin überwältigt von den Funktionen, die es zu bieten hat und freue mich in Zukunft mehr damit zu arbeiten.

Zudem möchte ich mich bei meinem Betreuer Simon Knaus für seine professionelle Hilfe bedanken. Ich schätze nicht nur seine offene und unkomplizierte Art, sondern auch, dass er mich selbstständig und mit vollstem Vertrauen an meinem Projekt arbeiten liess.

Bestätigung der Eigentätigkeit

Ich bestätige mit meiner Unterschrift, dass ich meine Maturaarbeit selbständig verfasst und in schriftliche Form gebracht habe, dass sich die Mitwirkung anderer Personen auf Beratung und Korrekturlesen beschränkt hat und dass alle verwendeten Unterlagen und Gewährspersonen aufgeführt sind. Mir ist bekannt, dass eine Maturaarbeit, die nachweislich ein Plagiat gemäss Art. 1quarter des Maturitätsprüfungsreglements des Gymnasiums (s. auch Maturaarbeitsbroschüre) darstellt, als schwerer Verstoss gewertet wird.

Datum und Unterschrift: .....

Literatur

[1] *The Quick, Draw! Dataset.* https://github.com/googlecreativelab/quickdraw-dataset. Abrufdatum: 7.11.2019.

[2] *Bresenham's Line Algorithm.* http://www.roguebasin.com/index.php?title=Bresenham%27s_Line_Algorithm, June 2018. Abrufdatum: 10.11.2019.

[3] *PyTorch Master Documentation.* https://pytorch.org/docs/stable/index.html, 2019. Abrufdatum: 28.12.2019.

[4] M. Andersson, A. Maja, and S. Hedar, *Sketch Classification With Neural Networks: A Comparative Study of CNN and RNN on the Quick, Draw! Data Set*, 2018.

[5] D. Becker, *Rectified Linear Units (ReLU) in Deep Learning.* https://www.kaggle.com/dansbecker/rectified-linear-units-relu-in-deep-learning. In: Kaggle. Abrufdatum: 15.12.2019.

[6] J. Bresenham, *A Linear Algorithm for Incremental Digital Display of Circular Arcs*, Communications of the ACM, 20 (1977), pp. 100–106.

[7] T. Domhan, J. T. Springenberg, and F. Hutter, *Speeding up Automatic Hyperparameter Optimization of Deep Neural Networks by Extrapolation of Learning Curves*, in Twenty-Fourth International Joint Conference on Artificial Intelligence, 2015.

[8] J. Friedman, T. Hastie, and R. Tibshirani, *The Elements of Statistical Learning*, vol. 1, Springer series in statistics New York, 2001.

[9] X. Glorot, A. Bordes, and Y. Bengio, *Deep Sparse Rectifier Neural Networks.* http://proceedings.mlr.press/v15/glorot11a/glorot11a.pdf. In: Proceedings of the Fourteenth International Conference on Artificial Intelligence and Statistics (= Proceedings of Machine Learning Research), 2011. Abrufdatum: 10.11.2019.

[10] K. Guo, J. WoMa, and E. Xu, *Quick, Draw! Doodle Recognition.*

[11] G. E. Hinton, N. Srivastava, A. Krizhevsky, I. Sutskever, and R. R. Salakhutdinov, *Improving Neural Networks by Preventing Co-Adaptation of Feature Detectors*, arXiv preprint arXiv:1207.0580, (2012).

[12] T. Hoeger, C. Dew, F. Pauls, and J. Wilson, *NDJSON - Newline Delimited JSON.* https://github.com/ndjson/ndjson-spec, July 2013. Abrufdatum: 7.11.2019.

[13] J. Jongejan, H. Rowley, T. Kawashima, J. Kim, and N. Fox-Gieg, *Quick, Draw!* https://quickdraw.withgoogle.com/. Abrufdatum: 7.11.2019.

[14] E. Kussul and T. Baidyk, *Improved Method of Handwritten Digit Recognition Tested on MNIST Database*, Image and Vision Computing, 22 (2004), pp. 971–981.

[15] L. Lu, Y. Shin, Y. Su, and G. E. Karniadakis, *Dying ReLU and Initialization: Theory and Numerical Examples*, arXiv preprint arXiv:1903.06733, (2019).

[16] J. McCarthy, *The Dartmouth Artificial Intelligence Conference.* `http://www.dartmouth.edu/~ai50/program.html`. Abrufdatum: 7.11.2019.

[17] J. D. Paola and R. A. Schowengerdt, *The Effect of Neural-Network Structure on a Multispectral Land-Use/Land-Cover Classification*, Photogrammetric Engineering and Remote Sensing, 63 (1997), pp. 535–544.

[18] N. Qian, *On the Momentum Term in Gradient Descent Learning Algorithms*, Neural networks, 12 (1999), pp. 145–151.

[19] I. Sutskever, J. Martens, G. Dahl, and G. Hinton, *On the Importance of Initialization and Momentum in Deep Learning*, in International conference on machine learning, 2013, pp. 1139–1147.

Abbildungsverzeichnis

Verzeichnis der Codes

A Konfusionsmatrix

	alarm clock	ant	apple	axe	basket	birthday cake	book	broom	butterfly	candle	diamond	eye	fireplace	flower	frying pan	hot air ballon	leaf	motorbike	octopus	pencil	rainbow	rifle	snake	sun	table	The Eiffel Tower	The Great Wall of China	The Mona Lisa	tooth-brush	tornado	umbrella	wine bottle	wine glass
alarm clock	**0.769**	0.009	0.024	0.001	0.017	0.013	0.017	0.001	0.004	0.002	0.002	0.02	0.023	0.008	0.009	0.001	0.011	0.007	0.003	0.0	0.002	0.002	0.006	0.015	0.001	0.001	0.009	0.007	0.003	0.009	0.001	0.001	0.001
ant	0.004	**0.758**	0.001	0.005	0.003	0.002	0.001	0.007	0.007	0.0	0.003	0.009	0.009	0.008	0.012	0.001	0.004	0.023	0.025	0.006	0.001	0.015	0.021	0.016	0.008	0.002	0.021	0.0	0.016	0.007	0.001	0.0	0.001
apple	0.028	0.004	**0.903**	0.001	0.002	0.006	0.001	0.001	0.005	0.006	0.002	0.001	0.002	0.001	0.006	0.0	0.003	0.004	0.0	0.0	0.001	0.0	0.004	0.003	0.0	0.001	0.004	0.0	0.002	0.001	0.0	0.008	0.0
axe	0.002	0.012	0.0	**0.836**	0.001	0.001	0.004	0.005	0.002	0.003	0.004	0.002	0.003	0.008	0.004	0.002	0.007	0.008	0.002	0.005	0.004	0.005	0.012	0.001	0.01	0.002	0.007	0.002	0.02	0.008	0.015	0.001	0.004
basket	0.014	0.007	0.003	0.003	**0.792**	0.017	0.016	0.004	0.001	0.001	0.007	0.008	0.026	0.004	0.005	0.007	0.002	0.008	0.012	0.001	0.001	0.003	0.002	0.002	0.005	0.0	0.01	0.007	0.008	0.024	0.001	0.001	0.001
birthday cake	0.012	0.003	0.01	0.002	0.017	**0.864**	0.01	0.002	0.002	0.007	0.0	0.009	0.017	0.001	0.001	0.001	0.002	0.005	0.002	0.0	0.0	0.002	0.001	0.002	0.002	0.006	0.009	0.003	0.002	0.003	0.0	0.005	0.0
book	0.007	0.005	0.001	0.003	0.007	0.004	**0.86**	0.003	0.009	0.002	0.004	0.006	0.026	0.001	0.004	0.0	0.005	0.005	0.002	0.001	0.001	0.001	0.001	0.003	0.002	0.0	0.006	0.025	0.003	0.004	0.0	0.001	0.0
broom	0.0	0.008	0.0	0.005	0.004	0.001	0.0	**0.886**	0.001	0.01	0.0	0.0	0.008	0.001	0.001	0.001	0.003	0.003	0.003	0.031	0.0	0.0	0.006	0.001	0.0	0.008	0.002	0.0	0.01	0.003	0.0	0.004	0.0
butterfly	0.014	0.028	0.003	0.003	0.004	0.005	0.026	0.0	**0.825**	0.002	0.004	0.014	0.013	0.016	0.003	0.001	0.007	0.008	0.008	0.0	0.001	0.0	0.0	0.004	0.0	0.001	0.001	0.004	0.002	0.002	0.0	0.0	0.0
candle	0.001	0.002	0.003	0.007	0.0	0.007	0.002	0.014	0.0	**0.89**	0.0	0.0	0.002	0.001	0.0	0.0	0.002	0.002	0.001	0.017	0.001	0.0	0.001	0.001	0.0	0.006	0.002	0.001	0.006	0.002	0.001	0.027	0.001
diamond	0.003	0.007	0.002	0.003	0.011	0.001	0.002	0.001	0.002	0.001	**0.889**	0.01	0.004	0.002	0.003	0.005	0.011	0.003	0.002	0.002	0.001	0.002	0.004	0.01	0.001	0.0	0.002	0.001	0.002	0.009	0.002	0.001	0.001
eye	0.017	0.013	0.002	0.0	0.003	0.008	0.001	0.002	0.007	0.0	0.002	**0.862**	0.018	0.002	0.007	0.001	0.006	0.006	0.002	0.0	0.0	0.003	0.003	0.014	0.0	0.0	0.003	0.003	0.004	0.01	0.0	0.0	0.0
fireplace	0.005	0.009	0.0	0.001	0.02	0.012	0.015	0.017	0.002	0.001	0.001	0.01	**0.776**	0.004	0.001	0.002	0.003	0.008	0.009	0.001	0.001	0.002	0.002	0.001	0.006	0.008	0.012	0.049	0.004	0.016	0.0	0.001	0.0
flower	0.006	0.012	0.001	0.013	0.007	0.002	0.002	0.003	0.024	0.003	0.002	0.004	0.006	**0.821**	0.001	0.003	0.017	0.004	0.012	0.001	0.0	0.0	0.003	0.007	0.001	0.0	0.0	0.002	0.017	0.019	0.005	0.0	0.002
frying pan	0.007	0.016	0.004	0.004	0.003	0.004	0.003	0.005	0.001	0.001	0.002	0.014	0.006	0.005	**0.792**	0.012	0.009	0.007	0.005	0.004	0.002	0.03	0.019	0.002	0.003	0.001	0.011	0.002	0.012	0.01	0.003	0.001	0.001
hot air balloon	0.001	0.003	0.0	0.004	0.009	0.001	0.002	0.002	0.001	0.001	0.007	0.002	0.003	0.004	0.001	**0.91**	0.003	0.001	0.003	0.001	0.0	0.0	0.001	0.0	0.002	0.0	0.001	0.002	0.0	0.016	0.003	0.001	0.011
leaf	0.014	0.015	0.002	0.01	0.003	0.002	0.006	0.002	0.005	0.002	0.017	0.011	0.008	0.026	0.018	0.004	**0.772**	0.005	0.007	0.009	0.006	0.002	0.012	0.007	0.001	0.002	0.005	0.006	0.006	0.011	0.002	0.003	0.001
motorbike	0.001	0.049	0.001	0.003	0.006	0.004	0.003	0.002	0.003	0.0	0.0	0.005	0.019	0.002	0.007	0.001	0.002	**0.813**	0.019	0.001	0.004	0.005	0.01	0.002	0.01	0.001	0.011	0.001	0.006	0.009	0.0	0.0	0.0
octopus	0.003	0.023	0.001	0.002	0.01	0.001	0.001	0.003	0.004	0.0	0.002	0.003	0.01	0.005	0.003	0.005	0.002	0.011	**0.873**	0.001	0.002	0.001	0.007	0.01	0.001	0.004	0.004	0.001	0.001	0.004	0.001	0.0	0.0
pencil	0.0	0.003	0.0	0.007	0.0	0.0	0.003	0.017	0.001	0.026	0.001	0.001	0.003	0.001	0.003	0.001	0.007	0.003	0.001	**0.803**	0.001	0.039	0.026	0.0	0.001	0.002	0.01	0.001	0.025	0.007	0.0	0.007	0.001
rainbow	0.0	0.002	0.0	0.001	0.001	0.0	0.0	0.001	0.0	0.0	0.0	0.003	0.002	0.0	0.002	0.0	0.001	0.005	0.004	0.002	**0.946**	0.005	0.006	0.001	0.002	0.001	0.005	0.002	0.0	0.003	0.001	0.0	0.0
rifle	0.001	0.015	0.0	0.007	0.003	0.001	0.003	0.004	0.001	0.0	0.002	0.006	0.008	0.0	0.017	0.0	0.002	0.012	0.003	0.017	0.006	**0.765**	0.029	0.001	0.013	0.0	0.038	0.001	0.035	0.007	0.001	0.0	0.001
snake	0.003	0.017	0.002	0.009	0.003	0.001	0.003	0.003	0.001	0.006	0.003	0.005	0.002	0.002	0.018	0.001	0.005	0.01	0.006	0.019	0.013	0.024	**0.773**	0.001	0.001	0.001	0.028	0.001	0.027	0.008	0.001	0.0	0.002
sun	0.009	0.015	0.001	0.004	0.002	0.001	0.004	0.0	0.002	0.0	0.003	0.008	0.003	0.004	0.004	0.001	0.005	0.002	0.014	0.0	0.001	0.0	0.003	**0.9**	0.001	0.002	0.002	0.001	0.003	0.002	0.001	0.001	0.0
table	0.002	0.011	0.0	0.01	0.003	0.003	0.007	0.001	0.0	0.0	0.003	0.002	0.007	0.001	0.001	0.0	0.002	0.013	0.004	0.001	0.003	0.033	0.002	0.002	**0.861**	0.0	0.021	0.002	0.003	0.001	0.001	0.0	0.0
The Eiffel Tower	0.001	0.002	0.0	0.002	0.001	0.002	0.002	0.017	0.0	0.008	0.0	0.0	0.009	0.0	0.0	0.0	0.002	0.005	0.009	0.005	0.001	0.0	0.002	0.001	0.0	**0.912**	0.001	0.002	0.003	0.005	0.0	0.004	0.001
The Great Wall of China	0.006	0.045	0.0	0.009	0.01	0.012	0.016	0.002	0.002	0.0	0.001	0.005	0.019	0.002	0.01	0.0	0.008	0.016	0.004	0.012	0.007	0.049	0.082	0.005	0.02	0.001	**0.589**	0.003	0.05	0.009	0.0	0.0	0.001
The Mona Lisa	0.004	0.004	0.001	0.003	0.006	0.002	0.02	0.0	0.002	0.001	0.0	0.003	0.054	0.005	0.001	0.004	0.004	0.002	0.004	0.0	0.002	0.0	0.001	0.001	0.002	0.003	0.004	**0.857**	0.003	0.006	0.001	0.001	0.0
toothbrush	0.001	0.005	0.0	0.019	0.001	0.001	0.0	0.006	0.0	0.005	0.0	0.001	0.002	0.002	0.005	0.0	0.001	0.003	0.0	0.019	0.0	0.037	0.014	0.001	0.001	0.001	0.016	0.0	**0.851**	0.007	0.0	0.0	0.0
tornado	0.003	0.003	0.0	0.004	0.007	0.0	0.0	0.003	0.0	0.003	0.004	0.007	0.007	0.01	0.001	0.004	0.005	0.003	0.001	0.004	0.001	0.004	0.004	0.001	0.0	0.001	0.003	0.001	0.009	**0.902**	0.001	0.0	0.002
umbrella	0.001	0.004	0.0	0.017	0.001	0.0	0.002	0.002	0.001	0.0	0.004	0.006	0.001	0.018	0.005	0.006	0.006	0.002	0.005	0.001	0.002	0.003	0.005	0.002	0.002	0.0	0.002	0.001	0.002	0.007	**0.888**	0.0	0.003
wine bottle	0.003	0.003	0.005	0.002	0.002	0.007	0.002	0.005	0.002	0.033	0.0	0.001	0.003	0.0	0.001	0.001	0.003	0.003	0.0	0.006	0.0	0.001	0.002	0.0	0.0	0.003	0.002	0.0	0.0	0.002	0.0	**0.901**	0.004
wine glass	0.002	0.003	0.001	0.009	0.005	0.002	0.004	0.006	0.001	0.003	0.003	0.001	0.001	0.003	0.002	0.017	0.003	0.006	0.001	0.003	0.0	0.001	0.007	0.001	0.001	0.002	0.004	0.001	0.003	0.012	0.001	0.009	**0.881**

Tabelle 1: Konfustionsmatrix für Validierungsdaten (Berechnungen entsprechen dem neuronalen Netz mit ID 6 in Abbildung 9)

B Quellcode: Compiler

```
#-----------------------------COMPILER PROGRAMM-----------------------------#
#                           Erstellt von Eric Ceglie
#---------------------------------------------------------------------------#
#
# Beschreibung:
#  - Erstellt aus den von Google zu verfuegung gestellten Rohdaten
#    einen fuer das Programm Training.py geeigneten Datensatz
#  - Rohdaten muessen im full_simplified_NAME.ndjson Format sein
#  - Rohdaten lassen auf der folgenden Seite Herunterladen:
#    https://console.cloud.google.com/storage/browser/quickdraw_dataset
#    /full/simplified
#  - Rohdaten muessen sich im durch data_raw_dir angegebenen Pfad befinden
#  - Erstellter Datensatz wird im durch data_compiled_dir angegebenen Pfad
#    abgespeichert
#  - Aufloesung der Bilder im Datensatz laesst sich beliebig einstellen
#
#---------------------------------------------------------------------------#


import torch
import ndjson
import random
import numpy as np
import os
import pickle
import cv2


#---------------VARIABLEN---------------#

data_amount = int(5*(10**5)) # Datenmaenge

img_dim = (32, 32) # Aufloesung der Bilder mit Format: (Breite, Hoehe)

# Testet, ob die angegebene data_amount fuer den PC nicht zu hoch ist (
    Fehlermeldung, wenn zu hoch)
data_inputs = torch.zeros(data_amount, 1, img_dim[0], img_dim[1])
del data_inputs

data_compiled_dir = "QD_data_compiled/" # Pfad des Ordners, indem die Datensaetze
    abgespeichert werden sollen
data_raw_dir = "QD_data_raw/" # Pfad des Ordners, indem die Rohdaten enthalten
    sind

data_raw = []
data_names = []

```

```
45
46 #---------------FUNKTIONEN---------------#
47
48 # Funktion zum Laden der Rohdaten definieren
49 def laden():
50
51     # globale Variabeln aufrufen
52     global data_names
53     global data_raw
54     global data_amount
55
56     # QD_Daten Ordner oeffnen und jede Datei mit zugehoerigem Namen (label) zum
       Datensatz hinzufuegen
57     files_num = len(next(os.walk(data_raw_dir))[2])         # Anzahl Dateien in
       data_raw_dir
58     print(2*"\n" + "Anzahl verschiedener Gegenstaende: " + str(files_num) + 2*"\n"
       )
59     with os.scandir(data_raw_dir) as Eintraege:
60         i = 1
61         for Eintrag in Eintraege:                           # Jeden Eintrag einzel
        durchgehen und zum Datensatz hinzufuegen
62             with open(data_raw_dir + Eintrag.name) as f:
63                 data_raw_new = ndjson.load(f)
64             f.close()
65
66             # Alle Bilder, die nicht erkannt wurden werden hier aussortiert
67             j = 0
68             while j < int(data_amount/files_num):
69                 if not data_raw_new[i]["recognized"]:
70                     del data_raw_new[i]
71                 else:
72                     j += 1
73             del j
74
75             # ueberschuessige Daten loeschen
76             if i != files_num:
77                 del data_raw_new[(int(data_amount/files_num)):]
78             else:
79                 del data_raw_new[(data_amount-len(data_raw)):]
80
81             data_raw += data_raw_new
82             del data_raw_new
83             print("Anzahl momentan vorhandener Bilder im Datensatz :\t", len(
       data_raw))
84             data_names.append(Eintrag.name.replace("full_simplified_","").replace(
       ".ndjson",""))
85             i += 1
86     del i
87     del files_num
```

```
88      del Eintraege
89      data_amount = len(data_raw)
90
91      # Datensatz zufaellig anordnen
92      random.shuffle(data_raw)
93
94      # im Datensatz vorhandene Gegenstaende ausgeben
95      print(2*"\n")
96      names_str = ""
97      for i in range(len(data_names)):
98          names_str += data_names[i]
99          if i != (len(data_names)-1):
100             names_str += ", "
101     print("Vorhandene Gegenstaende:\n" + names_str)
102     del names_str
103
104     print(2*"\n")
105
106 # Bresenham Line Algorithmus definieren
107 def bresenham_line(x1, y1, x2, y2):
108
109     deltax = x2 - x1
110     deltay = y2 - y1
111
112     steep = abs(deltay) > abs(deltax)
113
114     if steep:
115         x1, y1 = y1, x1
116         x2, y2 = y2, x2
117
118     swapped = False
119     if x1 > x2:
120         x1, x2 = x2, x1
121         y1, y2 = y2, y1
122         swapped = True
123
124     deltax = x2 - x1
125     deltay = y2 - y1
126
127     error = int(deltax / 2.0)
128     ystep = 1 if y1 < y2 else -1
129
130     y = y1
131     points = []
132     for x in range(x1, x2 + 1):
133         coord = (y, x) if steep else (x, y)
134         points.append(coord)
135         error -= abs(deltay)
136         if error < 0:
```

```
            y += ystep
            error += deltax

    if swapped:
        points.reverse()

    return points

# data_raw auf data_inputs & data_labels uebertragen und umschreiben (compiling)
def kompilieren():

    # Input & Label Tensoren erstellen (mit 0 gefuellt)
    global data_inputs
    global data_labels
    data_inputs = torch.zeros(data_amount, 1, img_dim[0], img_dim[1])
    data_labels = torch.zeros(data_amount, len(data_names))

    z = 1
    randomList = list(range(data_amount))
    random.shuffle(randomList)
    for i in randomList:
        data_inputs_new = torch.zeros(1, 256, 256)
        data_labels_new = torch.zeros(len(data_names))

        for stroke in data_raw[i]["drawing"]:
            x0 = stroke[0][0]
            y0 = stroke[1][0]

            for j in range(0, len(stroke[0])):
                x1 = stroke[0][j]
                y1 = stroke[1][j]
                line = bresenham_line(x0, y0, x1, y1) # für Resultat ohne Linie
    bresenham_line(x0, y0, x1, y1) ersetze mit: [(x1, y1)]
                for xp, yp in line:
                    data_inputs_new[0][yp][xp] = 1
                x0 = x1
                y0 = y1

        for k in range(len(data_names)):
            if data_raw[i]["word"] == data_names[k]:
                data_labels_new[k] = 1



        np_data_input = np.zeros((1, img_dim[1], img_dim[0]))
        img = data_inputs_new[0].numpy()
        img = cv2.resize(img, img_dim, interpolation=cv2.INTER_AREA)

        np_data_input[0] = img
```

```
185
186         data_inputs_new = torch.from_numpy(np_data_input).float()
187
188         data_inputs[i] = data_inputs_new
189         data_labels[i] = data_labels_new
190
191         # momentanen Stand der Kompilierung ausgeben (in %)
192         if z % (data_amount/100) == 0:
193             print(str(int(100*z/data_amount)) + "% kompiliert")
194         z += 1
195
196 # Funktion fuer Abspeichern der fertig kompilierten Dateien definieren
197 def speichern():
198     # Daten abspeichern
199     # fuer jeden Datensatz wird immer ein einzelner, neuer Ordner erstellt
200     folders_existing = next(os.walk(data_compiled_dir))[1]
201     folders_existing.sort(key=int)
202     new_folder = "0"
203     for i in range(len(folders_existing)):
204         if new_folder == folders_existing[i]:
205             new_folder = str(int(new_folder)+1)
206         else:
207             break
208     os.mkdir(data_compiled_dir + new_folder)
209
210     pickle.dump(data_inputs, open(data_compiled_dir + new_folder + "/data_inputs",
        "wb"))
211     pickle.dump(data_labels, open(data_compiled_dir + new_folder + "/data_labels",
        "wb"))
212     pickle.dump(data_names, open(data_compiled_dir + new_folder + "/data_names", "
        wb"))
213
214
215     # Beschreibung des Datensatzes erstellen (als .txt file)
216     data_names_txt = data_names[0]
217     for i in range(len(data_names)-1):
218         data_names_txt += ",\n\t\t\t\t" + data_names[i+1]
219
220     description = "Beschreibung des Datensatzes\n\n\nDatenmänge:\t\t\t" + str(len(
        data_inputs)) + "\n\nDatenauflösung:\t\t\t" + str(img_dim[0]) + " x " + str(
        img_dim[1]) + "\n\nenthaltene Gegenstände:\t\t" + data_names_txt
221
222     f = open(data_compiled_dir + new_folder + "/readme.txt", "w+")
223     f.write(description)
224     f.close()
225
226     print(30*"\n")
227     print("Alle Daten wurden erflogreich unter " + data_compiled_dir + new_folder
        + "/ abgespeichert!")
```

```
228
229
230 #---------------PROGRAMM---------------#
231
232 # Alle vorhandenen Rohdaten laden und auf benoetigte Anzahl Daten reduzieren
233 laden()
234
235 # Daten kompilieren und skalieren
236 kompilieren()
237
238 # Dateien unter data_compiled_dir abspeichern
239 speichern()
```

Code 5: Quellcode des Programms zur Verarbeitung der Rohdaten

C Quellcode: Training

```
#----------------------------TRAINING PROGRAMM----------------------------#
#                          Erstellt von Eric Ceglie
#-------------------------------------------------------------------------#
#
# Beschreibung:
#  - trainiert neuronale Netze auf beliebig grosse Datensaetze
#    (sofern der Computer ueber genuegend Leistung verfuegt)
#  - Datensaetze muessen vorher mit dem Compiler.py Programm erstellt werden
#    und muss sich im Ordner mit dem durch data_compiled_dir vorgegebenen
#    Pfad befinden
#  - Anzahl der verdeckten Schichten ist beliebig einstellbar und die Anzahl
#    der Neuronen in diesen Schichten laesst sich auch veraendern
#  - Parameter der trainierten Netze werden in einen automatisch erstellten
#    Ordner abgespeichert (Ordner befindet sich im Pfad, der mit nn_model_dir
#    angegeben wird)
#  - Zudem wird ein Graph, der die Entwicklung der Genauigkeit und des
#    Fehlerwerts erstellt und abgespeichert
#  - Es wird auch eine readme.txt Datei erstellt, die die wichtigsten
#    Informationen des Trainings und des trainierten neuralen Netzes enthält
#  - Bei bedarf lassen sich auch bereits trainierte neuronale Netze abrufen
#    und weiter trainieren
#
#-------------------------------------------------------------------------#


import torch
import torch.nn as nn
import torch.nn.functional as F
import torch.optim as optim
from torch.autograd import Variable
import matplotlib.pyplot as plt
import matplotlib.animation as animation
from matplotlib import style
import cv2
import numpy as np
import pickle
import time
import os
import ndjson
import random


#---------------VARIABLEN---------------#

start_time = time.time() # Variable, um später die Laufzeit zu berechnen

data_compiled_dir = "QD_data_compiled/" # Pfad des Ordners, der die Datensaetze
```

```
    enthaelt
dataset_number = "0" # Nummer des Datensatzes
nn_model_dir = "nn_models/" # Pfad des Ordners, indem das trainierte NN
    abgespeichert werden soll

load_net = False # Gibt an, ob ein bereits trainiertes Netz abgerufen werden soll
nn_model_load_number = "0" # Nummer des bereits trainierten Datensatzes
runtime_total = 0
if load_net:
    data_compiled_dir = nn_model_dir + nn_model_load_number + "/"
    dataset_number = "training_data"

layers_connection_factor = 0.5 # Faktor, mit dem die Neuronen mit den verdeckten
    Schichten abnehmen bzw. zunehmen sollen
hidden_layers_num = 3 # Anzahl der verdeckten Schichten (hidden_layers_num>=0)
learning_rate = 0.1 # Lernrate
learning_momentum = 0.9 # Impuls
batch_groesse = 10 # Batch Groesse
dropout = True # Gibt an, ob Dropout Schichten eingebaut werden sollen
dropout_rate = 0.4 # Gibt die Dropout Rate an (sofern Dropout vorhanden)
lr_scheduler = True # Gibt an, ob ein Lernraten Zerfall stattfinden soll
lr_stepsize = 3 # Gibt an, nach wievielen Epochen die Lernrate mit Faktor gamma
    abnehmen soll
gamma = 0.8 # Faktor, mit dem die Lernrate abnehmen soll (0<gamma<1)
data_open_amount = 1 # Gibt an, welche Menge des Datensatzes benutzt werden soll
    (0<data_open_amount<=1)
training_epochen = 1000 # Gibt die maximale Anzahl der Trainingsepochen an

epochForGraph = []
lossForGraph = []
accuracyForGraphOnTrainingData = []
accuracyForGraphOnValidationData = []

data_amount_train = None
data_amount_validation = None
data_amount_test = None
nn_model_number = None
img_dim_flat = None
data_inputs = None
data_labels = None
data_names = None
data_inputs_validation = None
data_labels_validation = None
data_inputs_train = None
data_labels_train = None
DNA = None
net = None


```

```
#---------------FUNKTIONEN---------------#

# Eine Trainingsepoche definieren
def training_epoche():

    global lossForGraph

    net.train()

    lossMomSum = 0
    randomList = list(range(int(data_amount_train/batch_groesse)))
    random.shuffle(randomList)

    j = 0
    for i in randomList:
        j += 1
        #if j % (data_amount_train/10) == 0:
         #   print(str(int(100*j/data_amount_train)) + "%")

        inputs = data_inputs_train[i]
        labels = data_labels_train[i]
        inputs = inputs.cuda()
        labels = labels.cuda()
        inputs = Variable(inputs)
        labels = Variable(labels)

        optimizer.zero_grad()

        outputs = net(inputs)
        labels = labels.long()
        labels = labels.float()

        loss = criterion(outputs, labels)
        loss.backward()
        optimizer.step()

        lossMomSum += loss.item()

    if lr_scheduler:
        scheduler.step()

    lossForGraph.append(lossMomSum / (data_amount_train/batch_groesse))


def laden():

    global data_amount_train
    global data_amount_validation
    global data_amount_test
```

```
141     global nn_model_number
142     global data_inputs
143     global data_labels
144     global data_names
145     global data_inputs_validation
146     global data_labels_validation
147     global data_inputs_train
148     global data_labels_train
149     global img_dim_flat
150     global epochForGraph
151     global lossForGraph
152     global accuracyForGraphOnTrainingData
153     global accuracyForGraphOnValidationData
154     global runtime_total
155 
156     # für jedes nn Model wird immer ein einzelner, neuer Ordner erstellt
157     folders_existing = next(os.walk(nn_model_dir))[1]
158     folders_existing.sort(key=float)
159     nn_model_number = "0"
160     for i in range(len(folders_existing)):
161         if nn_model_number == folders_existing[i]:
162             nn_model_number = str(int(nn_model_number)+1)
163         else:
164             break
165 
166     if load_net:
167         try:
168             runtime_total = pickle.load(open(data_compiled_dir + "/runtime_total",
     "rb"))
169         except:
170             pass
171 
172     # Daten öffnen
173     data_inputs = pickle.load(open(data_compiled_dir + dataset_number + "/
    data_inputs", "rb"))
174     data_labels = pickle.load(open(data_compiled_dir + dataset_number + "/
    data_labels", "rb"))
175     data_names = pickle.load(open(data_compiled_dir + dataset_number + "/
    data_names", "rb"))
176 
177     if not load_net:
178         data_inputs_train = data_inputs[:int(0.8*len(data_inputs))]
179         data_labels_train = data_labels[:int(0.8*len(data_inputs))]
180 
181         data_inputs_validation = data_inputs[int(0.8*len(data_inputs)):int(0.9*len
    (data_inputs))]
182         data_labels_validation = data_labels[int(0.8*len(data_inputs)):int(0.9*len
    (data_inputs))]
183 
```

```
184         data_inputs_test = data_inputs[int(0.9*len(data_inputs)):]
185         data_labels_test = data_labels[int(0.9*len(data_inputs)):]
186 
187     else:
188         data_inputs_train = data_inputs
189         data_labels_train = data_labels
190 
191         data_inputs_validation = pickle.load(open(data_compiled_dir + "
    validation_data"  + "/data_inputs", "rb"))
192         data_labels_validation = pickle.load(open(data_compiled_dir + "
    validation_data" + "/data_labels", "rb"))
193 
194         data_inputs_test = pickle.load(open(data_compiled_dir + "test_data"  + "/
    data_inputs", "rb"))
195         data_labels_test = pickle.load(open(data_compiled_dir + "test_data" + "/
    data_labels", "rb"))
196 
197         epochForGraph = pickle.load(open(data_compiled_dir + "graph/GraphEpoch", "
    rb"))
198         lossForGraph = pickle.load(open(data_compiled_dir + "graph/GrapchLoss", "
    rb"))
199         accuracyForGraphOnTrainingData = pickle.load(open(data_compiled_dir + "
    graph/GrapchAccuracyTrain", "rb"))
200         accuracyForGraphOnValidationData = pickle.load(open(data_compiled_dir + "
    graph/GraphAccuracyVal", "rb"))
201 
202     if data_open_amount != 1:
203         data_inputs_train = data_inputs_train[:int(data_open_amount*len(
    data_inputs_train))]
204         data_inputs_test = data_inputs_test[:int(data_open_amount*len(
    data_inputs_test))]
205         data_inputs_validation= data_inputs_validation[:int(data_open_amount*len(
    data_inputs_validation))]
206 
207     print("Anzahl Trainingsdaten geladen:\t\t", len(data_inputs_train))
208     print("Anzahl Validierungsdaten geladen:\t", len(data_inputs_validation))
209     print("Anzahl Testdaten geladen:\t\t\t", len(data_inputs_test))
210 
211     data_amount_train = len(data_inputs_train)
212     data_amount_validation = len(data_inputs_validation)
213     data_amount_test = len(data_inputs_test)
214 
215     img_dim = (len(data_inputs_train[0][0]), len(data_inputs_train[0][0][0]))
216     img_dim_flat = img_dim[0]*img_dim[1]
217     print("Daten wurden geladen")
218 
219     # Neuen Ordner fuer das nn Model erstellen und alle Daten dort nochmal
    abspeichern (fuer zukuenftige uebersicht)
220     os.mkdir(nn_model_dir + nn_model_number)
```

```
221     os.mkdir(nn_model_dir + nn_model_number + "/training_data")
222     os.mkdir(nn_model_dir + nn_model_number + "/validation_data")
223     os.mkdir(nn_model_dir + nn_model_number + "/test_data")
224 
225     pickle.dump(data_inputs_train, open(nn_model_dir + nn_model_number + "/
        training_data/data_inputs", "wb"))
226     pickle.dump(data_labels_train, open(nn_model_dir + nn_model_number + "/
        training_data/data_labels", "wb"))
227     pickle.dump(data_names, open(nn_model_dir + nn_model_number + "/training_data/
        data_names", "wb"))
228 
229     pickle.dump(data_inputs_validation, open(nn_model_dir + nn_model_number + "/
        validation_data/data_inputs", "wb"))
230     pickle.dump(data_labels_validation, open(nn_model_dir + nn_model_number + "/
        validation_data/data_labels", "wb"))
231     pickle.dump(data_names, open(nn_model_dir + nn_model_number + "/
        validation_data/data_names", "wb"))
232 
233     pickle.dump(data_inputs_test, open(nn_model_dir + nn_model_number + "/
        test_data/data_inputs", "wb"))
234     pickle.dump(data_labels_test, open(nn_model_dir + nn_model_number + "/
        test_data/data_labels", "wb"))
235     pickle.dump(data_names, open(nn_model_dir + nn_model_number + "/test_data/
        data_names", "wb"))
236 
237     os.mkdir(nn_model_dir + nn_model_number + "/graph")
238 
239     # Daten in einzelne Batches aufteilen
240     data_inputs_train_new = []
241     data_labels_train_new = []
242     for i in range(int(data_amount_train/batch_groesse)):
243         data_inputs_train_new.append(data_inputs_train[i*batch_groesse:(i+1)*
        batch_groesse])
244         data_labels_train_new.append(data_labels_train[i*batch_groesse:(i+1)*
        batch_groesse])
245     data_inputs_train = data_inputs_train_new
246     data_labels_train = data_labels_train_new
247 
248 
249 def net_eval_accuracy():
250 
251     global accuracyForGraphOnValidationData
252     global accuracyForGraphOnTrainingData
253 
254     net.eval()
255 
256     accuracy = []
257     for i in range(len(data_inputs_validation)):
258         inputs = Variable(data_inputs_validation[i].cuda())
```

```
259         outputs = net(inputs).detach().cpu().numpy()[0].tolist()
260 
261         target_guess = outputs.index(min(outputs, key=lambda x:abs(x-1)))
262         target_real = data_labels_validation[i].numpy().tolist().index(1)
263 
264         if target_real == target_guess:
265             accuracy.append(1)
266         else:
267             accuracy.append(0)
268     accuracy = sum(accuracy)/len(accuracy)
269     accuracyForGraphOnValidationData.append(accuracy)
270     del accuracy
271 
272     accuracy = []
273     for i in range(len(data_inputs_validation)):
274         b = i//batch_groesse
275         inputs = Variable(data_inputs_train[b][i%batch_groesse].cuda())
276         outputs = net(inputs).detach().cpu().numpy()[0].tolist()
277 
278         target_guess = outputs.index(min(outputs, key=lambda x:abs(x-1)))
279         target_real = data_labels_train[b][i%batch_groesse].numpy().tolist().index
    (1)
280 
281         if target_real == target_guess:
282             accuracy.append(1)
283         else:
284             accuracy.append(0)
285     accuracy = sum(accuracy)/len(accuracy)
286     accuracyForGraphOnTrainingData.append(accuracy)
287     del accuracy
288 
289 #---------------KLASSEN---------------#
290 
291 # Netz als Klasse erstellen
292 class Net(nn.Module):
293 
294     def __init__(self):
295         super(Net, self).__init__()
296         self.layers_connection_factor = layers_connection_factor
297         self.hidden_layers_num = hidden_layers_num
298         self.input_layer_size = img_dim_flat
299         self.output_layer_size = len(data_names)
300 
301         # berechnung der model "DNA" bzw. der Anzahl hidden Layer und der Anzahl
    Neuronen
302         self.DNA = [self.input_layer_size]
303         for i in range(self.hidden_layers_num):
304             self.DNA.append(int(self.input_layer_size*(self.
    layers_connection_factor**(i+1))))
```

```
305         self.DNA.append(self.output_layer_size)
306 
307         self.linear_layers = nn.ModuleList([])
308         for i in range(len(self.DNA)-1):
309             self.linear_layers.append(nn.Linear(self.DNA[i], self.DNA[i+1]))
310         print("Modelstruktur:\n", self.linear_layers)
311 
312         self.dropout = nn.Dropout(p=dropout_rate)
313 
314     def forward(self, x):
315         x = x.view(-1, img_dim_flat)
316 
317         for i in range(len(self.linear_layers)):
318 
319             if dropout:
320                 x = self.dropout(x)
321 
322             x = self.linear_layers[i](x)
323 
324             if i != len(self.linear_layers)-1:
325                 x = F.relu(x)
326         return x
327 
328 
329 #---------------TRAINING---------------#
330 
331 laden()
332 
333 # Netz erstellen und auf Grafikkarte übertragen
334 net = Net()
335 if load_net:
336     net.load_state_dict(torch.load(nn_model_dir + nn_model_load_number + "/model")
    )
337 net = net.cuda()
338 
339 # Funktion für "Loss" Ausgabe definieren (direkt aus Pytorch)
340 criterion = nn.MSELoss()
341 #criterion = nn.CrossEntropyLoss()
342 
343 # Funktion für Optimierung definieren (direkt aus Pytorch)
344 optimizer = optim.SGD(net.parameters(), lr=learning_rate, momentum=
    learning_momentum)
345 
346 # Funktion für lr decay definieren (direkt aus Pytorch)
347 scheduler = torch.optim.lr_scheduler.StepLR(optimizer, step_size=lr_stepsize,
    gamma=gamma)
348 
349 fig, ax1 = plt.subplots()
350 ax1.set_xlabel("Epoche")
```

```
ax1.set_ylabel("Loss Wert")
ax1.plot(epochForGraph, lossForGraph, label="Loss Wert", color="red")

ax2 = ax1.twinx()
ax2.set_ylabel("Genauigkeit")
ax2.plot(epochForGraph, accuracyForGraphOnTrainingData, label="Genauigkeit für
    Trainingsdaten", color="blue")
ax2.plot(epochForGraph, accuracyForGraphOnValidationData, label="Genauigkeit für
    Validierungsdaten", color="green")

ax1.legend(loc='lower left', bbox_to_anchor=(0.0, 1.01), borderaxespad=0, frameon=
    False)
ax2.legend(loc='lower right', bbox_to_anchor=(1.01, 1.01), borderaxespad=0,
    frameon=False)
fig.tight_layout()

# Netz trainieren
for epoche in range(training_epochen):
    training_epoche()

    net_eval_accuracy()

    if len(epochForGraph) == 0:
        epochForGraph.append(1)
    else:
        epochForGraph.append(epochForGraph[-1]+1)

    # momentanen Stand des Trainings ausgeben
    for param_group in optimizer.param_groups:
        current_lr = param_group['lr']
    print("Epoche: ", epochForGraph[-1], "\t\tLoss: ", round(lossForGraph[-1], 8),
     "\t\t\tGktT: ", accuracyForGraphOnTrainingData[-1], "\t\tGktV: ",
    accuracyForGraphOnValidationData[-1], "\t\t Lr: ", '{:.2e}'.format(current_lr)
    )

    # Model und Loss-Entwicklung zwischenspeichern
    torch.save(net.state_dict(), nn_model_dir + nn_model_number + "/model")

    ax1.plot(epochForGraph, lossForGraph, label="Loss Wert", color="red")
    ax2.plot(epochForGraph, accuracyForGraphOnTrainingData, label="Genauigkeit für
     Trainingsdaten", color="blue")
    ax2.plot(epochForGraph, accuracyForGraphOnValidationData, label="Genauigkeit f
    ür Validierungsdaten", color="green")

    plt.savefig(nn_model_dir + nn_model_number + "/model_Entwicklung.svg")
    plt.savefig(nn_model_dir + nn_model_number + "/model_Entwicklung.pdf")

    pickle.dump(net.DNA, open(nn_model_dir + nn_model_number + "/model_DNA", "wb")
    )
```

```
390     pickle.dump(epochForGraph, open(nn_model_dir + nn_model_number + "/graph/
        GraphEpoch", "wb"))
391     pickle.dump(lossForGraph, open(nn_model_dir + nn_model_number + "/graph/
        GrapchLoss", "wb"))
392     pickle.dump(accuracyForGraphOnTrainingData, open(nn_model_dir +
        nn_model_number + "/graph/GrapchAccuracyTrain", "wb"))
393     pickle.dump(accuracyForGraphOnValidationData, open(nn_model_dir +
        nn_model_number + "/graph/GraphAccuracyVal", "wb"))
394
395     runtime = runtime_total + (time.time() - start_time)
396
397     pickle.dump(runtime, open(nn_model_dir + nn_model_number + "/runtime_total", "
        wb"))
398
399     # redme.txt Datei erstellen
400     data_names_txt = data_names[0]
401     for i in range(len(data_names)-1):
402         data_names_txt += ",\n\t\t\t\t" + data_names[i+1]
403     description = "Beschreibung des Models und Trainingsdaten\n\n\n"
404     description += "Modelstruktur:\n" + str(net.linear_layers).replace("ModuleList
        (","")[:-1] + "\n\n"
405     description += "Benutzte Loss Funktion:\n\n\t" + str(criterion) + "\n\n"
406     description += "Benutzter Optimizer:\n\n\t" + str(optimizer) + "\n\n"
407     description += "Dropout:\t\t\t" + str(dropout) + "(" + str(dropout_rate) + ")"
         + "\n\n"
408     description += "lr_scheduler:\t\t\t" + str(lr_scheduler) + "\n\n"
409     description += "lr_stepsize:\t\t\t" + str(lr_stepsize) + "\n\n"
410     description += "gamma:\t\t\t\t" + str(gamma) + "\n\n"
411     description += "Model DNA:\t\t\t" + str(net.DNA) + "\n\n"
412     description += "Epochen abgeschlossen:\t\t" + str(epochForGraph[-1]) + "\n\n"
413     description += "Laufzeit:\t\t\t" + str(round(runtime//3600)) + ":" + str(round
        ((runtime//60)%60)) + ":" + str(round(runtime%60)) + "\n\n"
414     description += "Trainingsdaten Mänge:\t\t" + str(data_amount_train) + "\n\n"
415     description += "Validierungsdaten Mänge:\t" + str(data_amount_validation) + "\
        n\n"
416     description += "Testdaten Mänge:\t\t" + str(data_amount_test) + "\n\n"
417     description += "Batch Grösse:\t\t\t" + str(batch_groesse) + "\n\n"
418     description += "Benutzter Datensatz:\t\t" + str(dataset_number) + "\n\n"
419     description += "Datenauflösung:\t\t\t" + str(len(data_inputs_train[0][0][0]))
        + " x " + str(len(data_inputs_train[0][0][0][0])) + "\n\n"
420     description += "enthaltene Gegenstände:\t\t" + data_names_txt
421
422     f = open(nn_model_dir + nn_model_number + "/readme.txt", "w+")
423     f.write(description)
424     f.close()
425
426
427 print("Training erfolgreich beendet!")
```

Code 6: Quellcode des Programms für das Training eines neuronalen Netzes

D Quellcode: Berechnung einer Konfusionsmatrix

```
#-----------------------ACCURACY_EVALUATION PROGRAMM-----------------------#
#                          Erstellt von Eric Ceglie
#--------------------------------------------------------------------------#
#
# Beschreibung:
#  - Erstellt eine Konfusionsmatrix eines beliebigen neuronalen Netzes
#  - Kann entwerde fuer Trainings-, Test- oder Validierungsdaten
#    ausgewertet werden
#  - Ausgabe in From einer .txt Datei, die eine Tabelle und die
#    durchschnittliche Genauigkeit enthaelt
#
#--------------------------------------------------------------------------#


import torch
import torch.nn as nn
import torch.nn.functional as F
import torch.optim as optim
from torch.autograd import Variable
import matplotlib.pyplot as plt
import matplotlib.animation as animation
from matplotlib import style
import cv2
import numpy as np
import pickle
import imageio
import ndjson
import random
from tabulate import tabulate


#---------------VARIABLEN---------------#

nn_model_dir = "nn_models/" # Pfad des Ordners, indem sich die trainierten
    neuronalen Netze befinden
nn_model_number = "0" # Nummer des neuronalen Netzes

test_on = "validation_data" # Gibt an, fuer welche Daten eine Konfusionsmatrix
    erstellte werden soll
#(zur Auswahl stehen: "training_data", "test_data", "validation_data")

data_inputs = None
data_labels = None
data_names = None
DNA = None
net = None
Konfusionsmatrix = None
```

```
46
47 accuracy = {}
48
49
50 #---------------KLASSEN---------------#
51
52 # Netz als Klasse erstellen
53 class Net(nn.Module):
54
55     def __init__(self):
56         super(Net, self).__init__()
57         self.DNA = DNA
58         self.linear_layers = nn.ModuleList([])
59         for i in range(len(self.DNA)-1):
60             self.linear_layers.append(nn.Linear(self.DNA[i], self.DNA[i+1]))
61
62
63     def forward(self, x):
64         x = x.view(-1, self.DNA[0])
65
66         for i in range(len(self.linear_layers)):
67             x = self.linear_layers[i](x)
68             if i != len(self.linear_layers)-1:
69                 x = F.relu(x)
70         return x
71
72
73 #---------------FUNKTIONEN---------------#
74
75 # alle benoetigten Dateien laden
76 def laden():
77
78     # Daten laden
79     global data_inputs
80     global data_labels
81     global data_names
82
83     data_inputs = pickle.load(open(nn_model_dir + nn_model_number + "/" + test_on
    + "/data_inputs", "rb"))
84     data_labels = pickle.load(open(nn_model_dir + nn_model_number + "/" + test_on
    + "/data_labels", "rb"))
85     data_names = pickle.load(open(nn_model_dir + nn_model_number + "/" + test_on +
      "/data_names", "rb"))
86
87     if test_on == "training_data":
88         data_inputs = data_inputs[:int(0.125*len(data_inputs))]
89         data_labels = data_labels[:int(0.125*len(data_labels))]
90
91
```

```
92      # model DNA laden
93      global DNA
94      DNA = pickle.load(open(nn_model_dir + nn_model_number + "/model_DNA", "rb"))
95
96      # Netz definieren und trainierte Parameter laden
97      global net
98      net = Net()
99      net.load_state_dict(torch.load(nn_model_dir + nn_model_number + "/model")) #
        map_location=torch.device('cpu') wenn keine Grafikkarte vorhanden ist
100
101 def net_eval_data():
102     global Konfusionsmatrix
103
104     # Genauigkeiten berechnen
105     for i in range(len(data_names)):
106         accuracy.update({data_names[i]:{}})
107         for j in range(len(data_names)):
108             accuracy[data_names[i]].update({data_names[j]:[]})
109
110     data_amount = len(data_inputs)
111     for i in range(data_amount):
112         if i % (data_amount/10) == 0:
113             print(str(int(100*i/data_amount)) + "%")
114         outputs = net(data_inputs[i]).detach().numpy()[0].tolist()
115
116         target_guess = outputs.index(min(outputs, key=lambda x:abs(x-1)))
117         target_real = data_labels[i].numpy().tolist().index(1)
118
119         accuracy[data_names[target_real]][data_names[target_guess]].append(1)
120         for i in range(len(data_names)):
121             if data_names[i] != data_names[target_guess]:
122                 accuracy[data_names[target_real]][data_names[i]].append(0)
123
124     # Durchschnitte berechnen
125     for i in data_names:
126         for j in data_names:
127             accuracy[i][j] = sum(accuracy[i][j])/len(accuracy[i][j])
128
129     # Konfusionsmatrix erstellen
130     head = [""]
131     for i in range(len(data_names)):
132         head.append(data_names[i])
133
134     Zeilen = []
135     for i in range(len(data_names)):
136         Zeilen.append([])
137     i = 0
138
139     for Eintrag in accuracy:
```

```
140         Zeilen[i].append(Eintrag)
141         for name in data_names:
142             #acc = round(accuracy[Eintrag][name], 3)
143             acc = accuracy[Eintrag][name]
144 
145             # Alle Werte über x sollen in Latex fett dargestellt werden (bzw. die
    Diagonale soll fett dargestellt werden)
146             if accuracy[Eintrag][name] < 1.1:
147                 Zeilen[i].append(acc)
148             else:
149                 Zeilen[i].append(r"\textbf{" + str(acc) + "}")
150         i += 1
151         del acc
152     del i
153 
154     Konfusionsmatrix = tabulate(Zeilen, headers=head, tablefmt="plain") # https://
    pypi.org/project/tabulate/
155     #Konfusionsmatrix = tabulate(Zeilen, headers=head, tablefmt="latex_raw") # für
     Latex
156     print(Konfusionsmatrix)
157 
158 def speichern():
159 
160     description = ""
161     description += "Konfusionsmatrix der Künstlichen Intelligenz für " + test_on +
     " Bildern:\n\n\n"
162     description += Konfusionsmatrix
163     description += "\n\nDurchschnittliche Wahrscheinlichkeit der KI für eine
    korrekte Schätzung: "
164 
165     average = 0
166     for i in data_names:
167         for j in data_names:
168             if i == j:
169                 average += accuracy[i][j]
170     average = average / len(data_names)
171 
172     description += str(average)
173 
174     f = open(nn_model_dir + nn_model_number + "/accuracy_on_" + test_on + ".txt",
    "w+")
175     #f = open(nn_model_dir + nn_model_number + "/accuracy_on_" + test_on + "_latex
    .txt", "w+") # für Latex
176     f.write(description)
177     f.close()
178 
179 
180 #---------------PROGRAMM---------------#
181 
```

```
182 laden()
183
184 net_eval_data()
185
186 speichern()
```

Code 7: Quellcode des Programms zur Berechnung der Genauigkeiten eines bereits trainierten neuronalen Netzes (Ausgabe in From einer Konfusionsmatrix)

E Quellcode: Evaluierung von Daten

```
#------------------------DATA_EVALUATION PROGRAMM-------------------------#
#                          Erstellt von Eric Ceglie
#--------------------------------------------------------------------------#
#
# Beschreibung:
#  - Mit diesem Programm lassen sich mit bereits trainierten neuronalen
#    Netzen Daten evaluieren
#  - Es koennen entweder Validierungsdaten oder komplett neue Daten auswerten
#  - Komplett neue Daten sollten auf einer 256x256 grossen Leinwand
#    gezeichnet werden und es sollte ein Pinsel mit 1 Pixel Durchmesser
#    verwendet werden
#  - Ich empfehle das Programm paint.net zum erstellen von neuen Daten
#  - Fuer beste Ergebnisse, sollten neue Daten zudem so gross wie moeglich
#    gezeichnet werden
#  - Komplett neue Daten muessen als "myDrawing.png" im Ordner, wo sich auch
#    dieses Programm befindet abgespeichert werden
#
#--------------------------------------------------------------------------#


import torch
import torch.nn as nn
import torch.nn.functional as F
import torch.optim as optim
from torch.autograd import Variable
import matplotlib.pyplot as plt
from matplotlib import style
import cv2
import numpy as np
import pickle
import imageio
import time
import ndjson
import random


#---------------VARIABLEN---------------#

nn_model_dir = "nn_models/" # Gibt den Pfad an, indem sich die trainierten
    neuronalen Netze befinden
nn_model_number = "0" # Gibt die Nummer des neuronalen Netzes an
eval_new_data = True # Gibt an, ob komplett neue Daten evaluiert werden sollen


#---------------KLASSEN---------------#

# Netz als Klasse erstellen
```

```
47 class Net(nn.Module):
48
49     def __init__(self):
50         super(Net, self).__init__()
51         self.DNA = DNA
52         self.linear_layers = nn.ModuleList([])
53         for i in range(len(self.DNA)-1):
54             self.linear_layers.append(nn.Linear(self.DNA[i], self.DNA[i+1]))
55
56     def forward(self, x):
57         x = x.view(-1, self.DNA[0])
58
59         for i in range(len(self.linear_layers)):
60             x = self.linear_layers[i](x)
61             if i != len(self.linear_layers)-1:
62                 x = F.relu(x)
63         return x
64
65
66 #---------------FUNKTIONEN---------------#
67
68 def laden():
69
70     # Daten laden
71     global data_inputs
72     global data_labels
73     global data_names
74
75
76     data_inputs = pickle.load(open(nn_model_dir + nn_model_number + "/
       validation_data/data_inputs", "rb"))
77     data_labels = pickle.load(open(nn_model_dir + nn_model_number + "/
       validation_data/data_labels", "rb"))
78     data_names = pickle.load(open(nn_model_dir + nn_model_number + "/
       validation_data/data_names", "rb"))
79
80     # im Datensatz vorhandene Gegenstaende ausgeben
81     print(2*"\n")
82     names_str = ""
83     for i in range(len(data_names)):
84         names_str += data_names[i]
85         if i != (len(data_names)-1):
86             names_str += ",\n"
87     print("Vorhandene Gegenstaende:\n\n" + names_str)
88     del names_str
89
90     # model DNA laden
91     global DNA
92     DNA = pickle.load(open(nn_model_dir + nn_model_number + "/model_DNA", "rb"))
```

```
93
94      # Netz definieren und trainierte Parameter laden
95      global net
96      net = Net()
97      net.load_state_dict(torch.load(nn_model_dir + nn_model_number + "/model",
        map_location=torch.device('cpu'))) # map_location=torch.device('cpu') wenn
        keine Grafikkarte vorhanden ist
98
99  def net_eval_my_data():
100     plt.clf()
101
102     img = imageio.imread("myDrawing.png")
103     img = img[:,:,0]
104     img = img.astype('float64')
105
106     for i in range(len(img)):
107         for j in range(len(img[i])):
108             img[i][j] = 255 - img[i][j]
109             img[i][j] = img[i][j] / 255
110
111     inputs_dim = (len(data_inputs[0][0]), len(data_inputs[0][0][0]))
112     img = cv2.resize(img, (inputs_dim), interpolation=cv2.INTER_AREA)
113
114     img = torch.from_numpy(img)
115     img = img.float()
116     inputs = img
117     outputs = net(inputs).detach().numpy()[0].tolist()
118     target = outputs.index(min(outputs, key=lambda x:abs(x-1)))
119     #print(data_names)
120     #print(outputs)
121
122     TextString = "Meine Vermutung ist: " + data_names[target]
123     #TextString += "\n(Output Wert: " + str(outputs[target]) + ")"
124
125     props = dict(boxstyle='round', facecolor=(0.4,0.4,0.4) , alpha=0.9)
126     plt.text(2, -3, TextString, fontsize=12, color=(0.9,0.9,0.9),
127         verticalalignment='center_baseline', bbox=props)
128
129     plt.imshow(img, cmap="gray")
130     plt.axis("off")
131
132     plt.show(block=False)
133     plt.pause(1)
134
135 def net_eval_validation_data():
136     r = random.randint(0, len(data_inputs)-1)
137     img = data_inputs[r].numpy()
138
139     outputs = net(data_inputs[r]).detach().numpy()[0].tolist()
```

```
140     plt.imshow(img[0], cmap="gray")
141     plt.axis("off")
142 
143     target = outputs.index(min(outputs, key=lambda x:abs(x-1)))
144     #print(data_names)
145     #print(outputs)
146 
147     TextString = "Meine Vermutung ist: " + data_names[target]
148     #TextString += "\n(Output Wert: " + str(outputs[target]) + ")"
149     #print(TextString)
150 
151     props = dict(boxstyle='round', facecolor=(0.4,0.4,0.4) , alpha=0.9)
152     plt.text(2, -3, TextString, fontsize=12, color=(0.9,0.9,0.9),
153         verticalalignment='center_baseline', bbox=props)
154     plt.show()
155 
156 
157 #---------------PROGRAMM---------------#
158 
159 # Validierungsdaten und das neuronale Netz laden
160 laden()
161 
162 # selbst gezeichnete Daten testen
163 while eval_new_data:
164     net_eval_my_data()
165 
166 # beim Training benutzte Daten testen
167 while not eval_new_data:
168     net_eval_validation_data()
```

Code 8: Quellcode des Programms zur Evaluierung von beliebigen Daten