Design und Entwurf eines Captcha-Solvers

Praktikumsarbeit

im Studiengang
M. Sc. Elektro- und Informationstechnik
Schwerpunkt Kommunikations- und Informationstechnik

vorgelegt von

Daniel Kampert
Matr.-Nr.: 734590

am 14.01.2018
an der Hochschule Düsseldorf

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

Abkürzungsverzeichnis

CNN	Convolutional Neural Network
ReLU	Rectified Linear Unit
SGD	Stochastic Gradient Descent

1 Einleitung

Diese Praktikumsarbeit beschreibt die Entwicklung eines *Convolutional Neural Networks* (CNN) zur Auswertung von sogenannten Captchas[1] im Rahmen der Mastervorlesung „*Künstliche Intelligenz und Softcomputing*“.

Zur Bildverarbeitung soll auf die bewährte *OpenCV*[2] zurückgegriffen werden. Bei diesem Framework handelt es sich um eine freie Programmierbibliothek für verschiedene Programmiersprachen, die verschiedene, optimierte, Algorithmen für verschiedene Aufgaben im Umfeld der Bildverarbeitung und des maschinellen Lernens mitbringt.

Als Framework für das neuronale Netz wird die High-Level Python-Bibliothek *Keras* verwendet, die auf den Computing Bibliotheken *Tensorflow*, *CNTK* oder *Theano* aufbaut und es dem Anwender erlaubt sehr schnell, sehr effiziente und leistungsstarke neuronale Netze zu entwerfen und zu realisieren.

Für die Codeentwicklung wird die aktuellste Version von Visual Studio (VS2017) verwendet, welche die Entwicklung von Python-Projekten unterstützt und darüber hinaus über einen ausgezeichneten Debugger, Versionsverwaltungs- und Dokumentationstools (hier Git und Doxygen) verfügt.

Das komplette Projekt ist unter *https://github.com/Kampi* als Open Source Projekt einsehbar und zum Download verfügbar.

[1] Auch CAPTCHA (Engl. Completely Automated Public Turing test to tell Computers and Humans Apart)

[2] Open Source Computer Vision Library

2 Aufgabenstellung

Die Aufgabenstellung bestand darin, ein *Convolutional Neural Network* (CNN) derart zu trainieren, dass es in die Lage versetzt wird Captchas automatisch zu erkennen und auszuwerten. Als Zielsystem dieser Aufgabe wird eine Windowsplattform und als Programmiersprache wird eine aktuelle Version von Python 3 gewählt.

Die erste Aufgabe ist es, eine geeignete Datenbasis mit einer großen Anzahl Captchas zu beschaffen. Für eine Demoanwendung wird eine Datenbasis der Webseite Kaggle.com [1] verwendet. Anschließend wird eine Python-Anwendung entwickelt, die die Daten vorverarbeitet und ein neuronales Netz trainiert, welches daraufhin unbekannte Captchas lösen soll. Im letzten Schritt wird die entwickelte Anwendung getestet und die Ergebnisse analysiert.

Die Anwendung soll zudem so konstruiert werden, dass Sie erweitert werden kann um unbekannte Captchas, also welche, die nicht im Trainingsdatensatz enthalten sind, einzulesen und auszuwerten. Ein mögliches Anwendungsszenario dafür wäre die Liveeingabe eines Captchas vom Bildschirm (z. B. von Webseiten) um dieses zu lösen.

3 Installation der benötigten Pakete

Für die Ausführung des Programms werden die folgenden Pakete und Programme benötigt:

- Python (es empfiehlt sich die x64 Variante der Version 3.6.2)
- Git
- Tensorflow
- Keras
- OpenCV
- Matplotlib

Als erstes wird Python installiert. Bei der Installation sollte darauf geachtet werden, dass der Pfad des Python-Interpreters bei der Installation in die *PATH*-Variable eingetragen wird (siehe Abbildung 1).

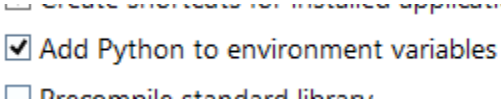

Abbildung 1: Der Pfad des Interpreters wird in die PATH-Variable eingetragen.

Für die Installation aller anderen Softwarepakete wird das Python-Modul *pip* verwendet, welches bei den Versionen > 3.4 automatisch mitgeliefert wird.

Über die Kommandozeile können die einzelnen Pakete installiert werden:

```
> python –m pip install numpy
> python –m pip install opencv-python
```

Für das Modul *Keras* werden noch einige zusätzliche Pakete benötigt:

```
> python –m pip install scipy
> python –m pip install scikit-learn
> python –m pip install pillow
> python –m pip install h5py
```

Als Backend für das Modul Keras wird noch *Tensorflow* oder *Theano* benötigt. Ich habe mich für das Backend *Tensorflow* entschieden, da Tensorflow, im Gegensatz zu Theano, über einen besseren Support und eine bessere Codepflege besitzt.

```
> python –m pip install tensorflow
```

Für große neuronale Netze empfiehlt sich die GPU basierte Version von Tensorflow, welche wie folgt installiert werden kann (siehe offizielle Dokumentation für die Nutzungsvoraussetzungen, wie unterstützte Grafikkarten etc.):

```
> python –m pip install tensorflow-gpu
```

Zu guter Letzt kann *Keras* installiert werden.

```
> python –m pip install git+git://github.com/fchollet/keras.git
```

Für eine Verarbeitung von Webseiten und die Erstellung von Plots werden noch die Module *requests* und *matplotlib* benötigt:

```
> python –m pip install requests
> python –m pip install matplotlib
```

Für die Visualisierung des Netzwerks werden die Module *graphviz* und *pydot* benötigt. Dazu muss die Executable von *graphviz* für das entsprechende Betriebssystem installiert und in die PATH-Variable eingetragen werden.

Dann können die Python-Module installiert werden:

```
> python –m pip install graphviz
> python –m pip install pydot
```

Das Einlesen von Captchas über den Bildschirm wird mit Hilfe der Module *pynput* und *pyautogui* realisiert:

```
> python –m pip install pyautogui
> python –m pip install pynput
```

Dieser Modus wird in dieser Dokumentation nicht explizit behandelt, aber für eine fehlerfreie Ausführung des Programms müssen die Pakete installiert werden.

Damit Visual Studio 2017 Python-Projekte unterstützen kann, müssen die Python-Tools installiert werden. Dazu wird Visual Studio geöffnet und unter *Neu* ▶ *Projekte* ▶ *Andere Sprachen* ▶ Python können die Tools dann installiert werden.

Ggf. muss noch die IntelliSense-Datenbank aktualisiert werden, damit die neu installierten Module nicht von der Codevervollständigung als unbekannte Module markiert werden.

Dazu wählt man im Projektmappen-Explorer den Eintrag *Python-Umgebungen* und anschließend über einen Rechtsklick den Menüpunkt *Alle Python-Umgebungen anzeigen* aus. Im Pull down-Menü der Python-Umgebung klickt man anschließend auf *IntelliSense* und dann auf *DB aktualisieren* (siehe Abbildung 2). Dieser Vorgang dauert für gewöhnlich etwas länger.

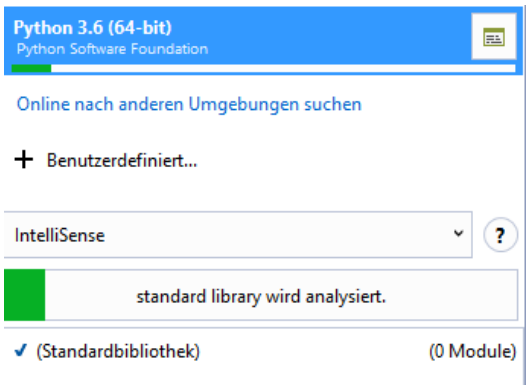

Abbildung 2: Die IntelliSense Datenbank wird aktualisiert

Die von Visual Studio verwendete Version von Python kann abschließend noch überprüft werden:

```
import sys
print (sys.version)
```

Sobald das Programm ausgeführt wird, muss die folgende Ausgabe erscheinen:

```
3.6.2 (v3.6.2:5fd33b5, Jul  8 2017, 04:57:36) [MSC v.1900 64 bit
(AMD64)]
Press any key to continue . . .
```

Damit ist die Entwicklungsumgebung fertig eingerichtet und alle notwendigen Python-Module sind installiert.

4 Trainingsdaten

Als Trainingsdaten für das Projekt wird ein Datensatz der Webseite Kaggle.com [1] verwendet. Dieser Datensatz enthält einige Captchas, die in Trainings- und Testdaten unterteilt sind (Abbildung 3).

Abbildung 3: Captcha-Datensatz von Kaggle.com

Ziel ist es, dass alle Captchas, die die Anwendung auswerten soll, eine identische Vorverarbeitung erhalten. Dies soll auch für Captchas gelten die in einer späteren Projektphase live vom Bildschirm aufgenommen werden (wie z. B. das Captcha aus Abbildung 4 von der Webseite der Telekom).

Abbildung 4: Captcha aus dem Internet - hier von der Telekom

Eine Schwierigkeit stellen Captchas dar, bei denen Linien oder sonstige Zeichen die Buchstaben bedecken, da diese die Bildverarbeitung davon abhalten die Buchstaben exakt zu erkennen und zu trennen. Solche Captchas sollen erst einmal vernachlässigt werden und dienen ggf. als Stoff für zukünftige Optimierungen.

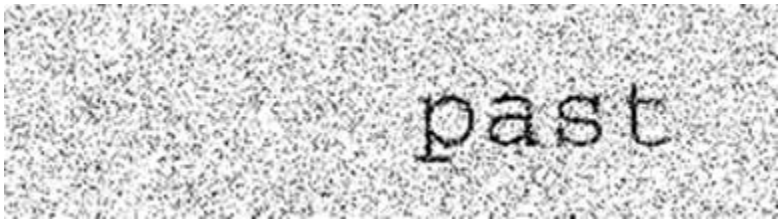

Abbildung 5: Mit Rauschen versehendes Captcha

Am Ende dieser Projektarbeit soll untersucht werden wie gut der entwickelte Ansatz bei verrauschten Captchas funktioniert (wie z. B. in Abbildung 5 zu sehen) und ob diese mit dem entwickelten Ansatz zuverlässig gelöst werden können.

5 Programmaufbau

Für die Realisierung werden verschiedene Modi (Abbildung 6) in das Programm, oder auch „*CaptchaBreaker*“, integriert:

- Einen Demo-Modus
  Dieser Modus verwendet die Captchas der Webseite *Kaggle.com*. Für das Training des Netzwerks werden die Captchas aus dem Verzeichnis *train* genutzt und eine Validierung des Netzwerks erfolgt mit den Captchas aus dem Verzeichnis *test*.
- Einen Live-Modus
  Der Live-Modus soll, nachdem das Netzwerk trainiert wurde, Captchas direkt vom Bildschirminhalt auslesen und auswerten (wird im Rahmen dieser Projektarbeit nicht implementiert).

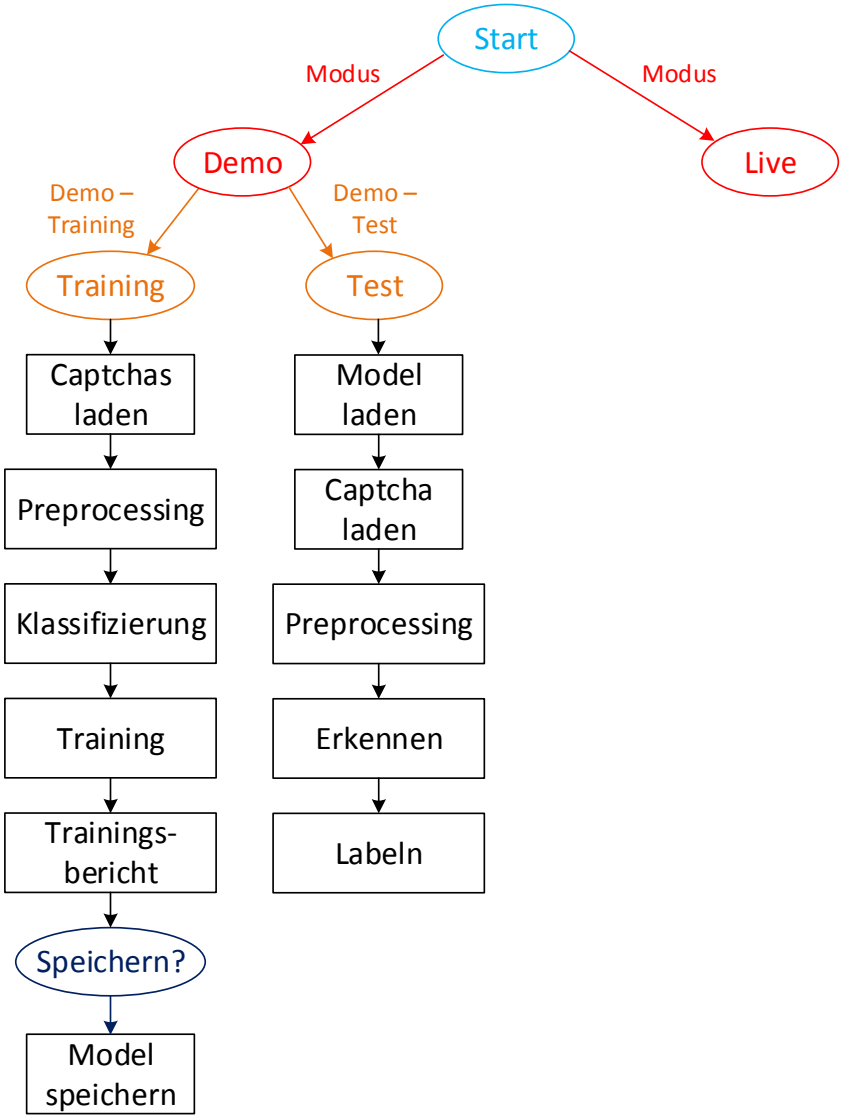

Abbildung 6: Schematischer Ablauf des Programms

Der Modus, sowie dessen Parameter (Tabelle 1), sollen vollständig über einen Kommandozeilenaufruf eingestellt werden können. Dazu wird das Python-Modul `argparse` verwendet.

	Funktion	Optionen
`-m`	Legt den Anwendungsmodus (hier nur `demo`) fest. `CaptchaBreaker.py -m demo`	`demo`, `live` (Default = `demo`)

`-w`	Legt fest, ob das Model trainiert oder geladen werden soll. Die Pfadangabe bestimmt von wo das Model geladen werden soll, bzw. wo das Datenverzeichnis für das Training liegt (absolute Pfade).
	`CaptchaBreaker.py -w load <PathToModel>`
	`CaptchaBreaker.py -w train <PathToFolder>`
		`train`, `load`

`-i`	Übergibt die auszuwertenden Captchas in das Programm. Hierbei können beliebig viele Captchas übergeben werden, welche dann nacheinander abgearbeitet werden (absolute Pfade).
	Ist der erste Eintrag ein Ordner, so werden automatisch alle Captchas aus diesem Ordner ausgewertet.
	`CaptchaBreaker.py -i <File1> <File2> … <FileN>`
	`CaptchaBreaker.py -i <Folder>`	

`-d`	Bestimmt ob eine Doku (Trainingsreport, Trainingsplot und ein Model des neuronalen Netzes) erstellt werden soll. `CaptchaBreaker.py -d <OutputDir> <ReportFile> <TrainingPlotFile> <ModelFile>`	
`-s`	Bestimmt, ob das neuronale Netz bei Programmende gespeichert werden soll (Optional). `CaptchaBreaker.py -s <OutputDir>`	

Tabelle 1: Kommandozeilenparameter des Beispielprogramms

5.1 Feature extraction

Sämtliche Captchas, egal ob sie zum Trainieren des neuronalen Netzes oder zur Auswertung verwendet werden, müssen vorverarbeitet werden, damit das CNN die Daten korrekt analysieren kann. Die Vorverarbeitung übernimmt die Klasse `ImagePreprocessing` des Moduls `Preprocessor` im `CaptchaSolver`.

Bei der Vorverarbeitung wird das Captcha geladen und in die einzelnen Zeichen zerlegt. Diese Zeichen werden, im Fall eines Trainings, in entsprechenden Verzeichnissen abgelegt bzw. im Fall einer Captcha Analyse werden die einzelnen Zeichen direkt in das CNN gegeben, welches die Zeichen dann auswertet.

Ziel der Vorverarbeitung mit OpenCV ist es, ein komplettes Captcha (Beispielhaft in Abbildung 7 zu sehen)

Abbildung 7: Das komplette Captcha

in einzelnen Zeichen zu zerlegen (siehe Abbildung 8), welche dann anschließend gespeichert werden können.

Abbildung 8: Ausgabe der Bildverarbeitung

Dabei unterteilt sich die Verarbeitung des Captchas in folgende Schritte (Abbildung 9).

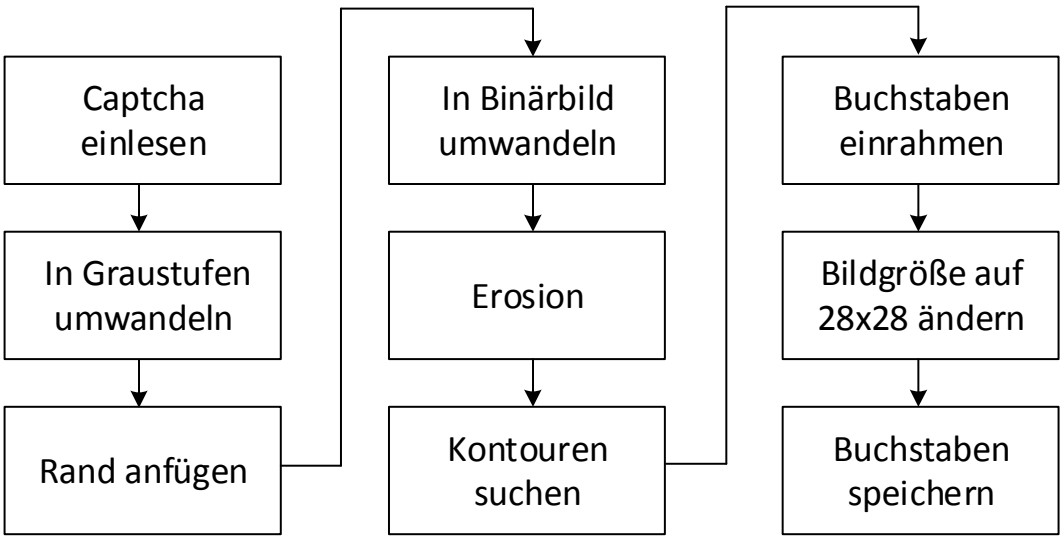

Abbildung 9: Vorverarbeitung des Captchas

Die Schwierigkeit bei dem verwendeten Demodatensatz, und mit sehr großer Wahrscheinlichkeit auch bei anderen Captchas, besteht darin, dass eine große Anzahl der Captchas über Buchstaben verfügt, die sehr dicht aneinander liegen. Dies ist bei den Buchstaben „U“ und „N“ in Abbildung 7 besonders gut zu sehen.

Um das Problem zu lösen wird das erzeugte Binärbild mit einer Erosion bearbeitet. Dies führt dazu, dass die einzelnen Zeichen dünner und der Abstand zwischen den Zeichen größer wird (siehe Abbildung 10).

Abbildung 10: Auswirkung der Erosion (rechts) auf das Binärbild (links)

Als Kernel für die Erosion wurde ein 3x3 Kreuz gewählt:

```
Kernel =  cv2.getStructuringElement(cv2.MORPH_CROSS, (3, 3))
…
BinaryImage = cv2.erode(BinaryImage, Kernel, iterations = 1)
```

Die Erosion entspricht dabei einer 2D-Faltung des Bildes mit einem Kernel, wie er z. B. nachfolgend definiert ist.

$$K = \begin{pmatrix} 0 & 1 & 0 \\ 1 & 1 & 1 \\ 0 & 1 & 0 \end{pmatrix}$$

Im letzten Schritt wird das Bild nach Konturen abgesucht, die dann dazu genutzt werden um die Objekte einzurahmen.

```
[im2, Contours, Hierarchy] = cv2.findContours(BinaryImage.copy(),
cv2.RETR_EXTERNAL, cv2.CHAIN_APPROX_SIMPLE)

for Contour in Contours:
   (x, y, w, h) = cv2.boundingRect(Contour)
   ROI = BinaryImage[y:y + h, x:x + w]
   ROI = cv2.resize(ROI, (self.ImageWidth, self.ImageHeight))
   cv2.imshow("Preview", ROI)
```

Der erzeugte Bildausschnitt wird anschließend auf eine einheitliche Größe, hier 28x28 Pixel, geändert, damit dieser in das CNN gegeben werden kann.

5.2 Klassifizierung der Trainingsdaten

Nach der Zerlegung des Captchas werden die einzelnen Buchstaben über die Methode `LoadTrainingData` der Klasse `CaptchaSolver` im Modul `CaptchaSolver` klassifiziert. Dieser Prozess wird vom Anwender des Programms einmalig durchgeführt. Während der *Feature extraction* werden die einzelnen Buchstaben angezeigt (siehe Abbildung 11) und der Anwender teilt dem Programm durch eine Tastatureingabe mit, welcher Buchstabe angezeigt wird.

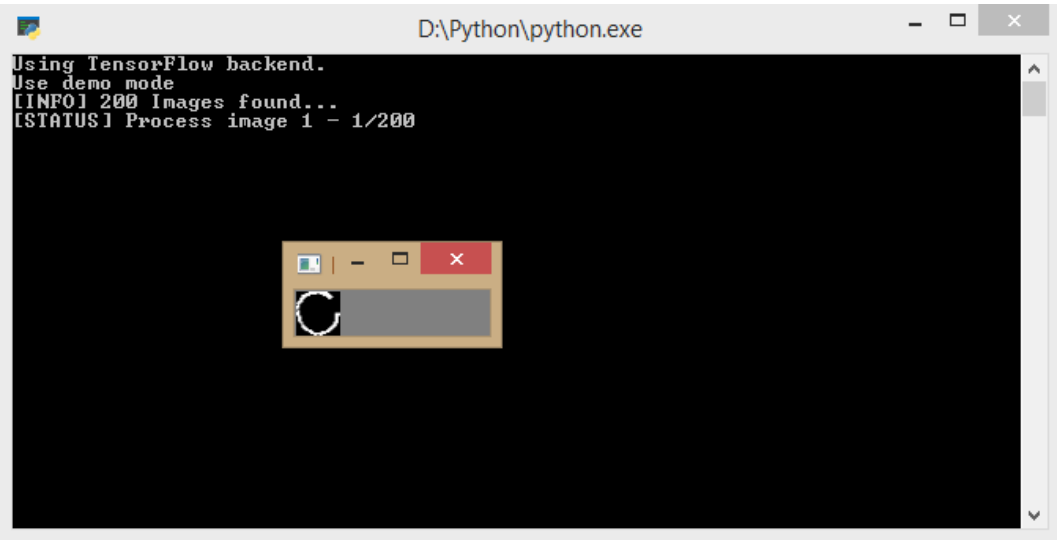

Abbildung 11: Das Programm wartet auf eine Eingabe vom Nutzer

Der angezeigte Buchstabe wird dann als PNG-Datei unter dem angegebenen Verzeichnis in einem gleichnamigen Verzeichnis gespeichert (siehe Abbildung 12).

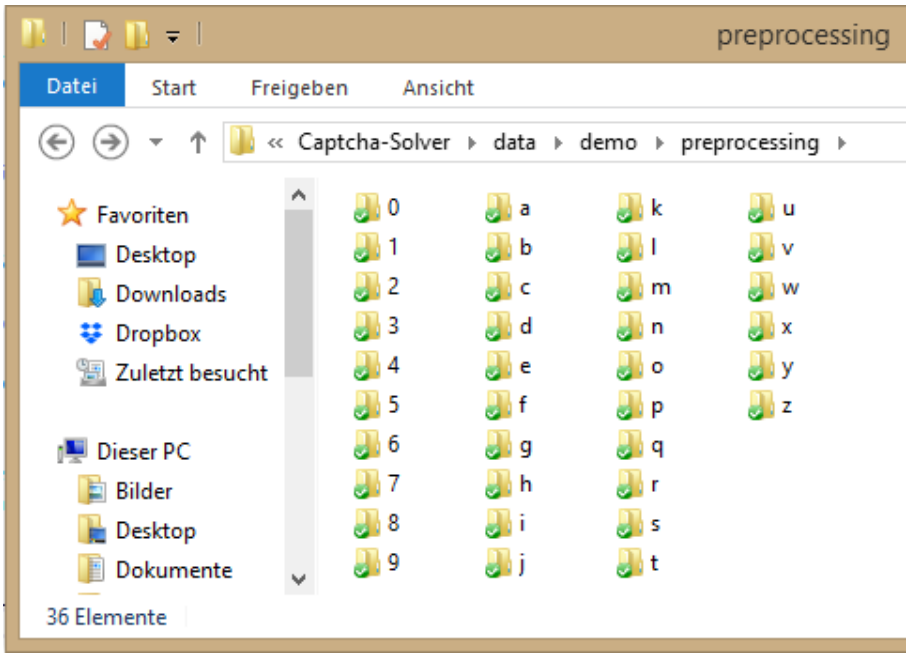

Abbildung 12: Ordnerstruktur der Trainingsdaten

Jedes Verzeichnis enthält dann verschiedene Bilder des entsprechenden Buchstaben (siehe Abbildung 13).

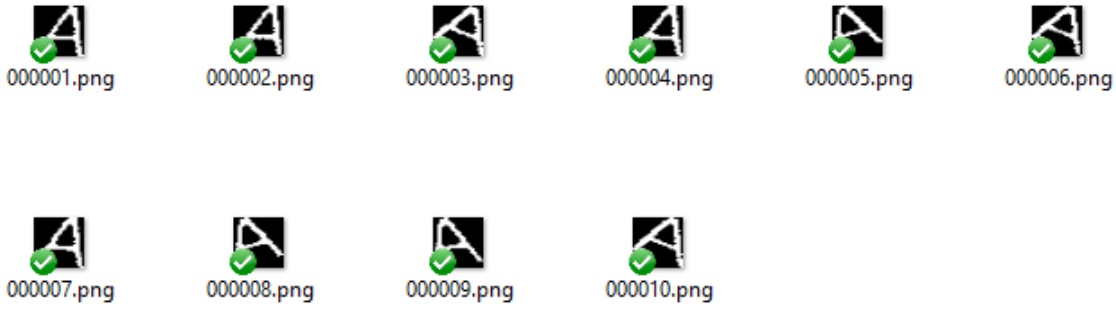

Abbildung 13: Samples des Buchstaben "A"

Sobald das Netz trainiert wird, werden die Bilder aus den einzelnen Unterordnern geladen, in ein numpy-Array konvertiert und in einem Array gespeichert. Das Array wird anschließend normiert um eventuelle Ausreißer bei einzelnen Pixeln zu unterdrücken.

```
Image = cv2.imread(OutputPath + "\\" + FolderName + "\\" + FileName,0)
Image = img_to_array(Image)
self.TrainingData.append(Image)
self.TrainingLabel.append(FolderName)
…
self.TrainingData = numpy.array(self.TrainingData, dtype = "float") /
255.0
```

Für jedes Bild wird ein entsprechendes Label, welches aus dem Namen des Bildverzeichnisses abgeleitet wird, gespeichert. Diese Label werden im Anschluss daran durch einen *Label Binarizer* in Vektoren für das neuronale Netz umgewandelt.

```
self.TrainingLabel.append(FolderName)
…
Binarizer = LabelBinarizer().fit(self.TrainingLabel)
self.TrainingLabel = Binarizer.transform(self.TrainingLabel)
```

Beispielhaft resultiert für das Label „a“ und 15 verschiedene Label der Vektor $(1 \quad 0 \quad \dots \quad 0 \quad 0)^T$. Die Länge des Vektors richtet sich nach der Menge der eingelesenen Label, wobei sich die Menge der eingelesenen Label an der Anzahl der klassifizierten Buchstaben orientiert. Enthalten beispielsweise nur 15 der 36 Unterverzeichnisse Trainingsdaten für die jeweiligen Buchstaben, so besitzt der Ergebnisvektor des *Label Binarizer* die Länge 15 (für jedes Label eine Spalte).

5.3 Design des neuronalen Netzes

Als Architektur für das neuronale Netz wurde *LeNet*[3] gewählt (Abbildung 14) und in der Klasse `LeNet` des Moduls `LeNet` implementiert. Die Stärke dieser Architektur liegt in der Handschrifterkennung. Da Captchas eine (optisch) sehr starke Ähnlichkeit mit Handschriften besitzen, stellt diese Architektur die ideale Wahl für diese Aufgabe dar.

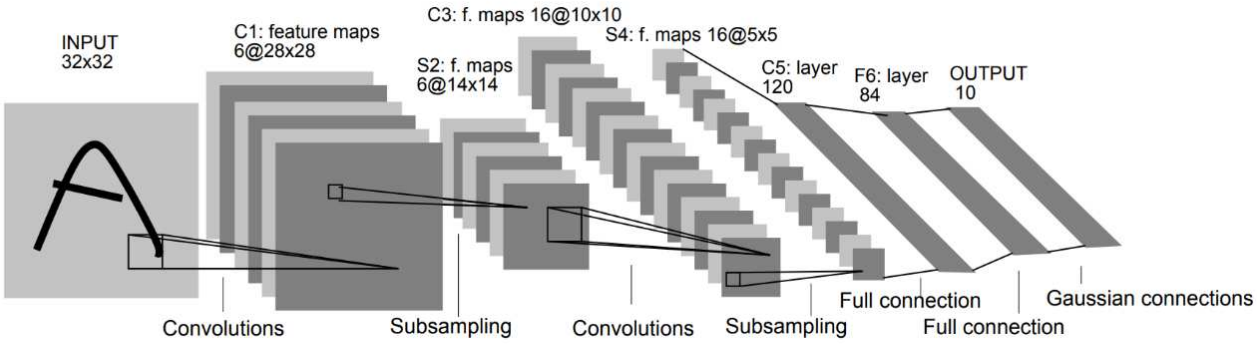

Abbildung 14: Architektur des LeNet [2]

Die Architektur verwendet somit den folgenden Aufbau:

INPUT → CONV → ACT → POOL → CONV → ACT → POOL →FC → ACT → FC → SOFT

Kürzel	**Funktion**
INPUT	Eingang
CONV	Filter
ACT	Aktivierungsfunktion (hier $tanh$[4])
POOL	Pooling
FC	Fully connect
SOFTMAX	Softmax-Funktion

Tabelle 2: Layerbezeichnungen

Das Netzwerk wird fast komplett so übernommen, wie es beschrieben wird. Einzig die Aktivierungsfunktionen und die Parameter werden geändert. Als neue Aktivierungsfunktion ersetzt die $ReLU$ den $tanh$.

[3] Siehe Paper „Gradient-Based Learning Applied to Document Recognition" von LeCun et al. aus dem Jahre 1998

[4] Tangens hyperbolicus

Die ReLU, (siehe Abbildung 15) definiert als

$$f(x) = max(0, x)$$

wurde 2000 von Hahnloser et al. in dem Paper „*Digital selection and analogue amplification coexist in a cortex-inspired silicon circuit*“ eingeführt und gilt als besonders effizient was die Implementierung in einem digitalen System angeht [3].

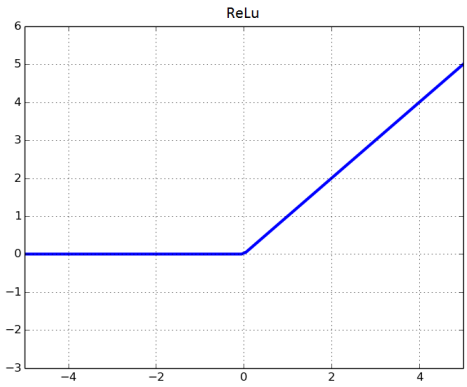

Abbildung 15: Die ReLU-Funktion

Hahnloser und Seung haben in ihrem Paper „*Permitted and Forbidden Sets in Symmetric Threshold-Linear Networks*“ aus dem Jahr 2003 zudem nachgewiesen, dass die ReLU Funktion den natürlichen Lernprozess eher nachbildet als andere Aktivierungsfunktionen [3].

Die Parameter der LeNet Architektur werden dabei folgendermaßen definiert:

Layer	**Ausgang**	**Filtergröße, -schrittweite**
INPUT	28 x 28 x 1	
CONV	28 x 28 x 20	5 x 5, K = 20
ACT	28 x 28 x 20	
POOL	14 x 14 x 20	2 x 2
CONV	14 x 14 x 50	5 x 5, K = 20
ACT	14 x 14 x 50	
POOL	7 x 7 x 50	2 x 2
FC	500	
ACT	500	
FC	36	
SOFTMAX	36	

Tabelle 3: Parameter der LeNet Architektur

5.4 Die Klasse CaptchaSolver

Die Klasse *CaptchaSolver* stellt die eigentliche Implementierung der Captcha-Auswertung, bestehend aus dem *LeNet* und dem *Preprocessor*, dar. Nachfolgend soll die Struktur des CaptchaSolvers erörtert werden.

5.4.1 __init__(int Width, int Height, int Epochs, int Depth, int Batchsize)

Diese Methode stellt den Konstruktor der Klasse dar.

```
__init__(self, Width, Height, Epochs = 30, Depth = 1, Batchsize = 32,
Bordersize = 8)
```

Argument	Beschreibung
`self`	Verweis auf die Klasse selbst
`Width`	Breite der zu verarbeitenden Captcha-Buchstaben (Default = `28`)
`Height`	Höhe der zu verarbeitenden Captcha-Buchstaben (Default = `28`)
`Epochs`	Anzahl der Trainingsdurchläufe (Default = `30`)
`Depth`	Farbkanäle der Captcha-Buchstaben (Default = `1`)
`Batchsize`	Größe des Verarbeitungsstapels (Default = `32`)
`Bordersize`	Rahmen, welcher vor der Verarbeitung an jeder Seite hinzugefügt wird (Default = `8`)

5.4.2 __del__()

Diese Methode stellt den Dekonstruktor der Klasse dar.

```
__del__(self)
```

Argument	Beschreibung
`self`	Verweis auf die Klasse selbst

5.4.3 SetDebugOption(bool DebugStatus)

Mit dieser Methode werden Debug-Ausgaben beim Lernen und bei der Vorhersage aktiviert.

```
SetDebugOption(self, DebugStatus)
```

Argument	Beschreibung
`self`	Verweis auf die Klasse selbst
`DebugStatus`	Boolescher Wert. Aktiviert oder deaktiviert die Debug-Ausgabe.

5.4.4 PrintModel(string OutputPath, string ModelName)

Mit dieser Methode kann das trainierte Netzwerk als Bilddatei gespeichert werden.

```
PrintModel(self, OutputPath, string ModelName = "Model.png")
```

Argument	Beschreibung
`self`	Verweis auf die Klasse selbst
`OutputPath`	Speicherort des Bildes
`ModelName`	Name der Bilddatei (Default = `Model.png`)
`return`	Fehlercode

5.4.5 Report(string OutputPath, string ReportName, string PlotName)

Über diese Methode kann ein Report über das letzte Training abgerufen und gespeichert werden. Wenn der Ausgabepfad nicht leer ist, wird der Report lokal gespeichert.

```
Report(self, OutputPath = "", ReportName = "Report.txt", PlotName = "Plot.png")
```

Argument	Beschreibung
`self`	Verweis auf die Klasse selbst
`OutputPath`	Ausgabepfad des Reports (Default = `""`)
`ReportName`	Name der Reportdatei (Default = `Report.txt`)
`PlotName`	Name der Bilddatei mit dem Plot (Default = `Plot.png`)
`return`	Fehlercode

5.4.6 LoadModel(string InputPath, string ModelFileName, string LabelfileName)

Mit dieser Methode kann ein abgespeichertes LeNet von einem lokalen Datenträger aus einer HDF5[5]-Datei geladen werden. Gleichzeitig werden die Zähler für die Erkennungsrate zurückgesetzt.

Es werden zwei Dateien eingelesen, welche sich in einem einzelnen Verzeichnis befinden müssen:

- Das Model
- Ein Labelvektor

```
LoadModel(self, InputPath, ModelFileName = "Model ", LabelFileName = "Label")
```

[5] Hierarchial Data Format: Datenformat für wissenschaftliche Anwendungen

Argument	Beschreibung
`self`	Verweis auf die Klasse selbst
`InputPath`	Verzeichnis mit den benötigten Dateien
`ModelFileName`	Name der Datei mit dem Model (Default = `Model`)
`LabelFileName`	Name der Datei mit den Labeln (Default = `Label`)
`return`	Fehlercode

5.4.7 SaveModel(string OutputPath, string ModelFileName, string LabelfileName)

Mit dieser Methode kann das trainierte LeNet im HDF5-Format abgespeichert werden. Dabei werden zwei Dateien erzeugt:

- Das Model
- Eine Datei mit dem Labelvektor

```
SaveModel(self, OutputPath, ModelFileName = "Model", LabelFileName = "Label")
```

Argument	Beschreibung
`self`	Verweis auf die Klasse selbst
`OutputPath`	Ausgabeverzeichnis
`ModelFileName`	Name der Datei mit dem Model (Default = `Model`)
`LabelFileName`	Name der Datei mit den Labeln (Default = `Label`)
`return`	Fehlercode

5.4.8 LoadTrainingData(string InputPath, string OutputPath, double SplitRatio, int RandomState)

Lädt neue Trainingsdaten für das Model und startet die Vorverarbeitung der Daten.

```
LoadTrainingData(self, InputPath, OutputPath, SplitRatio = 0.25, RandomState = 0)
```

Argument	Beschreibung
`self`	Verweis auf die Klasse selbst
`InputPath`	Verzeichnis mit den Trainingscaptchas
`OutputPath`	Zielverzeichnis für die verarbeiteten und klassifizierten Daten

`SplitRatio`	Teilungsverhältnis zwischen Trainings- und Testdaten für das Training der eingelesenen Daten (Default = `0.25`)
`RandomState`	Seed-Wert für den Zufallsgenerator der die eingelesenen Daten in Trainings- und Testdaten aufteilt (Default = `0`)
`return`	Fehlercode

5.4.9 TrainModel()

Trainiert das LeNet mit den, durch `LoadTrainingData()`, eingelesenen Daten.

```
TrainModel(self)
```

Argument	Beschreibung
`self`	Verweis auf die Klasse selbst
`return`	Fehlercode

5.4.10 ResetCounter()

Setzt die Zähler für die korrekten Vorhersagen zurück.

```
ResetCounter(self)
```

Argument	Beschreibung
`self`	Verweis auf die Klasse selbst

5.4.11 GetCounter()

Gibt die aktuellen Zählerstände für korrekte Vorhersagen zurück.

```
GetCounter(self)
```

Argument	Beschreibung
`self`	Verweis auf die Klasse selbst
`return`	[Erkannte Zeichen, Korrekt erkannte Buchstaben]

5.4.12 Predict(string InputImagePath, int[] DrawingColor, bool Debug)

Trifft eine Vorhersage für das übergebene Captcha. Das Captcha wird dabei eingelesen, in seine Buchstaben unterteilt und jeder Buchstabe wird analysiert und angezeigt.

```
Predict(self, InputImagePath, DrawingColor = (0, 0, 255), Debug = False)
```

Argument	Beschreibung
`self`	Verweis auf die Klasse selbst

`InputImagePath`	Pfad des zu analysierenden Captchas
`DrawingColor`	Farbe, mit der die Captchas markiert und gelabelt werden Default = `(0, 0, 255)`
`Debug`	Vorschau und händische Bestätigung aktivieren bzw. deaktivieren Default = `False`
`return`	Bild mit erkannten Captcha oder Fehlercode

6 Evaluierung

In diesem Kapitel möchte ich auf die ersten Versuche mit der entwickelten Software, der Bildverarbeitung und dem neuronalen Netz eingehen. Die ersten Versuche dienten dazu, die Grenzen der Bildverarbeitung und den dadurch vorhandenen Einfluss auf das neuronale Netz herauszufinden und um die Fähigkeiten des neuronalen Netzes zu testen.

Im Anschluss daran wird die Anwendung bei verrauschten Captchas getestet um zu überprüfen wie zuverlässig die Anwendung bei solchen Captchas arbeitet.

6.1 Der erste Test

Im ersten Schritt der Evaluierung wurde die Anwendung im Demo-Modus gestartet und trainiert.

```
> python CaptchaBreaker.py –m demo –w train D:\Machine-
Learning\CaptchaBreaker\data –i D:\Machine-
Learning\CaptchaBreaker\data\test\900
```

Im Demo-Modus werden folgende Funktionen genutzt (siehe Tabelle 4).

Methode	**Klasse**	**Beschreibung**
`CaptchaSolver`	`CaptchaSolver`	Konstruktor
`LoadTrainingData`	`CaptchaSolver`	Laden der Trainingsdaten und erstellen der Trainings- und Testdaten für das Training
`TrainModel`	`CaptchaSolver`	Trainiert das neuronale Netz mit den Trainingsdaten der Methode `LoadTrainingData`
`Predict`	`CaptchaSolver`	Trifft eine Vorhersage anhand eines Bildes
`SaveModel`	`CaptchaSolver`	Speichert das Model auf der Festplatte

Tabelle 4: Ablauf des Demo-Modus

Für das erste Training wurden 60 der 200 Captchas aus dem Verzeichnis *train* ausgewertet, klassifiziert und die Ergebnisse gespeichert.

Dabei hat sich herausgestellt, dass die Erosion viele Probleme mit zusammenhängenden Buchstaben löst, aber auch einige Probleme, wie ein abgeschnittenes „R“, erzeugt (siehe Abbildung 16).

Abbildung 16: Durch die Erosion abgeschnittenes „R“

Auch scheint die Erosion bei einige zusammenhängenden Buchstaben nicht zu funktionieren (siehe Abbildung 17). Während das „R“ aus Abbildung 16 für die Buchstaben trotzdem verwendet werden kann, gilt dies nicht für die Buchstaben aus Abbildung 17. Diese müssen verworfen werden.

Abbildung 17: Trotz Erosion hängen diese beiden Buchstaben immer noch zusammen

Nach der Klassifizierung durch den Anwender wird das Netzwerk in 30 Durchläufen trainiert. Abbildung 18 zeigt den Verlauf der Fehler-Funktion, sowie die Genauigkeit des Trainings während der einzelnen Durchläufe.

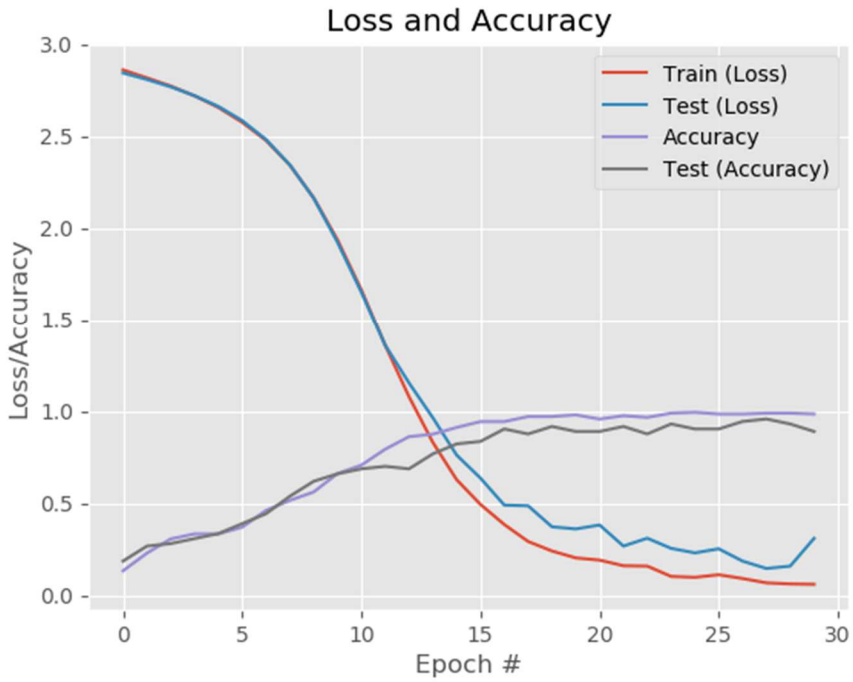

Abbildung 18: Verlust und Genauigkeit des Netzwerks nach dem Training

Für das Training werden die klassifizierten Bilder und die Label verwendet, welche durch eine `train_test_split`-Funktion in Trainings- und Testdaten unterteilt werden. Anhand der Trainings- und der Testdaten ermittelt das neuronale Netz während des Trainings den aktuellen Fehler und korrigiert mittels SGD die gelernten Werte, um den Fehler zu minimieren.

Die Ergebnisse eines durchgeführten Trainings können der Tabelle 5 entnommen werden.

	Precision	Recall	F1-Score	Support
a	1.00	1.00	1.00	4

b	1.00	1.00	1.00	5
c	1.00	1.00	1.00	3
e	0.00	0.00	0.00	6
f	1.00	0.33	1.00	3
g	0.50	1.00	0.67	6
h	1.00	1.00	1.00	2
j	1.00	1.00	1.00	9
k	1.00	1.00	1.00	3
l	1.00	0.86	1.00	3
m	1.00	1.00	0.80	2
n	1.00	0.75	0.86	8
p	0.45	1.00	0.62	5
r	1.00	0.40	0.57	5
t	1.00	1.00	1.00	4
u	1.00	1.00	1.00	2
x	1.00	1.00	1.00	2
y	1.00	0.25	1.00	2
avg / total	0.91	0.89	0.88	74

Tabelle 5: Trainingsreport

Nach dem Training kann das gelernte Model abgespeichert, oder direkt zum Auswerten von Captchas verwendet werden (Default Modus).

Für eine erste Vorhersage wurde ein einzelnes Captcha aus dem *test*-Verzeichnis ausgewählt und in die `Predict`-Methode übergeben:

```
Solver.Predict(args["input"])
```

Das angegebene Captcha wird nun auf dieselbe Art vorverarbeitet wie die Trainingsdaten und anschließend in die `predict`-Methode des Models übergeben.

Das Ergebnis der Prädiktion ist ein Vektor (hier der Übersichtlichkeit halber in Matrixform dargestellt) mit Erkennungswahrscheinlichkeiten für das eingelesene Zeichen auf Grundlage der trainierten Zeichen. Als Beispiel das Ergebnis für den Buchstaben „L“ aus dem Captcha *test/900*.

$$P = \begin{pmatrix} 1.444e^{-6} & 2.983e^{-3} & 2.0354e^{-3} & 1.5389e^{-2} & 4.3358e^{-5} & 2.8558e^{-5} \\ 3.4093e^{-5} & 1.0170e^{-5} & 1.5282e^{-3} & 9.6066e^{-1} & 1.6303e^{-4} & 1.0205e^{-4} \\ 5.5033e^{-5} & 1.4375e^{-4} & 2.3762e^{-7} & 1.6730e^{-2} & 1.0313e^{-5} & 7.4462e^{-5} \end{pmatrix}$$

In diesem Vektor wird dann nach dem höchsten Wert gesucht (rot markiert) und der Index des höchsten Wertes (hier 9) stellt dann den Index des erkannten Zeichens im Labelvektor des Trainings dar.

$$L = (a \quad b \quad c \quad e \quad f \quad g \quad h \quad j \quad k \quad l \quad m \quad n \quad p \quad r \quad t \quad u \quad x \quad y)^T$$

Mit Hilfe des Prädiktionsergebnisses wird der Labelvektor ausgelesen und der erkannte Buchstabe in der Konsole ausgegeben:

```
print("Label: {}".format(self.Binarizer.classes_[Prediction.argmax(axis = 1)]))
```

Zusätzlich wird der gerade ausgewertete Buchstabe auch noch angezeigt. So kann der Nutzer direkt schauen ob die Auswertung korrekt war (siehe Abbildung 19).

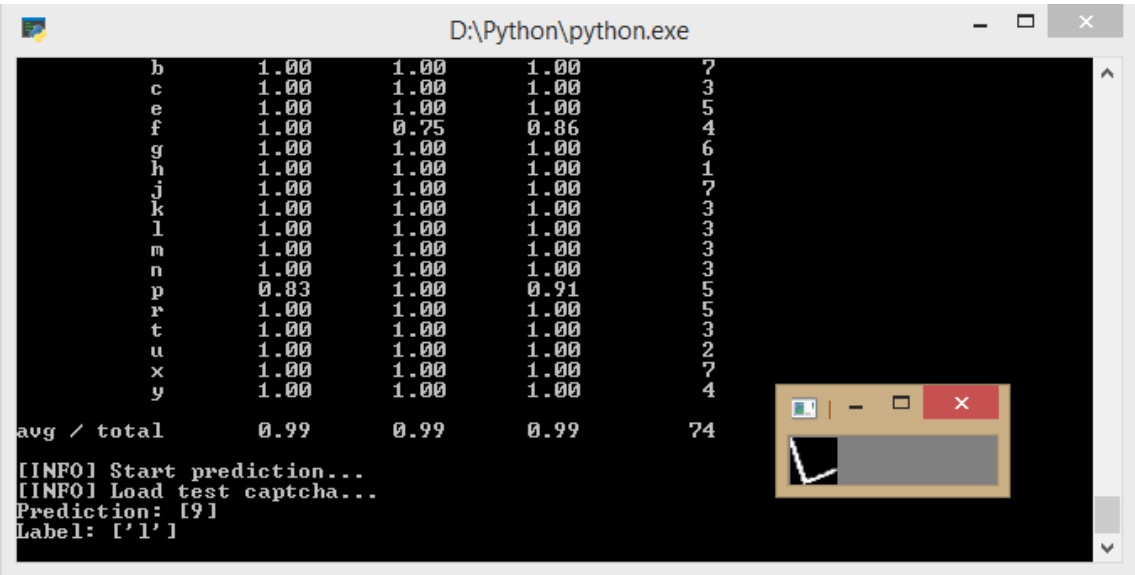

Abbildung 19: Ergebnis einer Prädiktion

Für weitere Tests wird das Model am Ende der Prädiktion lokal gespeichert. Dadurch werden alle Tests mit dem gleichen neuronalen Netz durchgeführt, da bei einem Training die Trainingsdaten zufällig aufgeteilt werden und somit bei jedem Training die gelernten Werte zufällig variieren.

6.2 Einfluss von Rauschen auf die Auswertung

Nun sollen die Captchas mit Rauschen versehen werden. Damit soll getestet werden, wie gut das neuronale Netz bei verrauschten Captchas im Internet (siehe Abbildung 20) anwendbar ist.

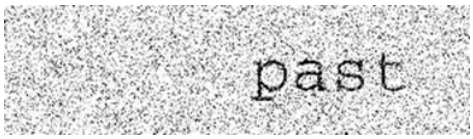

Abbildung 20: Ein verrauschtes Captcha aus dem Internet

Mit Hilfe von Matlab wurden die Captchas eingelesen, mit Gaußschen Rauschen versehen und anschließend gespeichert.

```
ImageAfter = imnoise(ImageBefore, 'gaussian', Mean, Variance);
```

Da Matlab keine Bilder ohne Dateiendung unterstützt, wurde der Name eines jeden Captchas bei der Ausgabe in <Name>.jpg geändert. Dies hat allerdings keine Auswirkung auf die Bearbeitung, da die Captchas vor der Bearbeitung durch Matlab ebenfalls JPG-Dateien waren.

Über die Parameter `Mean` und `Variance` (in dB) kann der Einfluss des Rauschens eingestellt werden (beispielhaft in Abbildung 21 zu sehen).

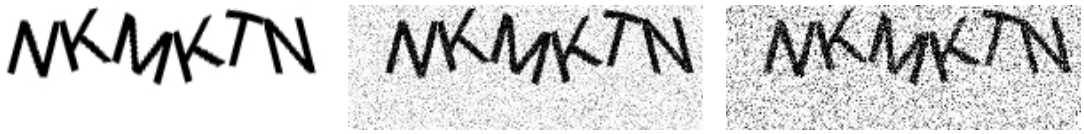

Abbildung 21: Captcha bei einem SNR von 100 dB (Links), 5 dB (Mitte) und 0.01 dB (Rechts) und einem Mean von 0

Es zeigt sich gut, dass das Captcha aus Abbildung 20 mit einem SNR von etwa 0.01 dB versehen ist.

Für die Auswertung wurden die folgenden Bilder verwendet:

- 545
- 572
- 596
- 660
- 709
- 869
- 900
- 918
- 936
- 951

Die veränderten Captchas werden nun nacheinander in den *CaptchaBreaker* geladen und die Erkennungsrate ausgewertet.

Es haben sich bei einer händischen Auswertung für ein SNR von 100 dB, 5 dB und 0.01 dB folgende Erkennungsraten ergeben (Tabelle 6).

SNR	Captchas	Erkannte Zeichen	Erkennungsrate
100 dB	10	39/54	72,2%
5 dB	10	43/69	62,3%
0.01 dB	10	21/197	10,6%

Tabelle 6: Erkennungsraten bei verschiedenen SNR-Werten

Durch Tabelle 6 wird sehr gut sichtbar, dass ein höheres Rauschen für eine Zunahme der erkannten Zeichen durch die Vorverarbeitung führt (siehe Abbildung 22).

Abbildung 22: Verrauschtes Binärbild für die Auswertung

Diese Zeichen sind oft Bruchstücke der eigentlichen Buchstaben, welche durch die Erosion und das Rauschen entstanden sind (siehe Abbildung 23 links). Auf der anderen Seite werden einige Buchstaben, trotz des starken Rauschens, korrekt erkannt (siehe Abbildung 23 rechts).

Abbildung 23: Ein, durch Rauschen und die Erosion, entstandenes Fragment eines Buchstaben (Links) und ein korrekt erkannter Buchstabe (Rechts)

Die Auswertung zeigt ganz deutlich, dass die bisher angewandte Vorverarbeitung der Captchas nicht verwendet werden kann, um ein Captcha, wie es in Abbildung 20 gezeigt ist, zuverlässig zu lösen. Dieses Problem kann ggf. durch eine darauf optimierte Bildverarbeitung gelöst werden. Der jetzige Ansatz kann dies aber definitiv nicht leisten.

6.3 Probleme mit fremden Captchas

Zum Schluss wurde überprüft wie gut sich der *CaptchaBreaker* auf neue, im Internet heruntergeladene, Captchas anwenden lässt. Als Basis wurden Captchas von der Webseite der deutschen Telekom verwendet (siehe z. B. in Abbildung 24).

Abbildung 24: Captcha der deutschen Telekom

6.3.1 Beschaffung der benötigten Datenbasis

Dazu wird einige Captchas benötigt. Die Seite der Telekom bietet den Vorteil, dass die Captchas generiert und als Bild eingebettet werden. Somit können die Bilder automatisiert heruntergeladen werden. Bei den heruntergeladenen Bildern handelt es sich allerdings um GIFs, welche OpenCV nicht öffnen kann. Die Bilder müssen deswegen in JPG-Bilder konvertiert werden.

```
import io
import time
import imghdr
import argparse
import urllib.request
from PIL import Image

CaptchaURL = "…"
ProjectDir = "…"

ArgParser = argparse.ArgumentParser()
ArgParser.add_argument("-c", "--count", help = "Number of captchas for
download", required = True)
args = vars(ArgParser.parse_args())

for Captcha in range(0, int(args["count"])):
    Path = ProjectDir + "data\\download\\" + str(Captcha)
    Response = urllib.request.urlopen(CaptchaURL)
    Request = Response.read()
    Type = imghdr.what(io.BytesIO(Request))
    if(Type == "gif"):
        print("[INFO] Found gif image. Start conversion...")
        Img = Image.open(io.BytesIO(Request))
        RGB = Img.convert("RGB")
        RGB.save(Path + ".jpg")
    print("[INFO] Save captcha {} / {}".format(Captcha + 1,
int(args["count"])))
    time.sleep(0.5)
```

Mit Hilfe des Programms werden nun 120 Captchas der Telekom heruntergeladen und im Verzeichnis *download* gespeichert.

```
> python CaptchaBreaker.py –c 120
```

6.3.2 Auswertung der Captchas

Nach dem Download werden die Captchas in Trainings- und in Testdaten unterteilt. Mit diesen Trainingsdaten wird dann das Netzwerk trainiert.

```
> python CaptchaBreaker.py –m demo –w train D:\Dropbox\GitHub\Machine-
Learning\CaptchaBreaker\data\download –i D:\Dropbox\GitHub\Machine-
Learning\CaptchaBreaker\data\download\test
```

Bei der Verarbeitung der Captchas hat sich herausgestellt, dass die Captchas der Telekom nicht verarbeitet werden konnten. Es hat sich herausgestellt, dass das Problem hier in der Vorverarbeitung der Bilder lag.

Die Captchas von Kaggle.com wurden durch das Einlesen und den Threshold in ein schwarzes Bild mit weißer Schrift umgewandelt. Das Captcha der Telekom wird durch die selbe Vorverarbeitung in ein weißes Bild mit schwarzer Schrift umgewandelt (siehe Abbildung 25).

Abbildung 25: Unterschiedliche Ergebnisse der Vorverarbeitung

Die anschließende Erosion sorgt dafür, dass die weißen Stellen im Bild schmaler werden. Bei dem Bild links in Abbildung 25 führt dies zum gewünschten Ergebnis. Bei dem rechten Bild allerdings sorgt die Erosion dafür, dass die Schrift größer wird (siehe Abbildung 26).

Abbildung 26: Falsches Ergebnis der Erosion

Die komplette Bildbearbeitung ist auf ein Szenario mit einem schwarzen Bild und einer weißen Schrift ausgelegt. Für die Captchas der Telekom muss eine Dilatation mit anschließender Invertierung stattfinden oder das Bild muss invertiert und dann mittels Erosion bearbeitet werden. In beiden Fällen muss die Vorverarbeitung angepasst werden.

Es wurde eine Mittelwertberechnung der Pixelintensität eingefügt um zu bestimmen, ob es sich um ein schwarzes oder ein weißes Bild handelt. Abhängig vom Ergebnis wird das Bild invertiert.

```
if(cv2.mean(BinaryImage)[0] > 128.0):
   print("[INFO] Detect white image...")
   cv2.bitwise_not(BinaryImage, BinaryImage)
```

Die invertierten Bilder werden anschließend weiterverarbeitet.

Bei den anschließenden Tests hat sich herausgestellt, dass die Erkennungsrate bei diesen Captchas etwa gleich hoch ist wie bei den Captchas von Kaggle.com. Ein Training des Netzwerks mit den Buchstaben der Captchas von Kaggle.com und der Telekom hat es dem Netzwerk zudem ermöglicht Captchas aus beiden Quellen zuverlässig zu erkennen.

7 Zusammenfassung und Fazit

Dieses Projekt war ein guter Einstieg in die Bildverarbeitung, das maschinelle Lernen und die, mit diesen beiden Themen verbundenen, Herausforderungen.

Das Ziel, ein Programm zu entwickeln, welches mittels eines neuronalen Netzes Captchas löst, wurde erreicht. Während der Entwicklungs- und Evaluierungsphase wurde festgestellt, dass eine gute Vorverarbeitung der Captchas mitunter der wichtigste Teil in der Anwendung darstellt, da das neuronale Netzwerk mit diesen Daten lernt und aus diesen Daten Vorhersagen trifft. Fehler in den Daten führen unweigerlich zu einem falschen Lernen oder zu einer ungenauen Prädiktion.

Es hat sich gezeigt, dass die aktuelle Vorverarbeitung noch nicht in der Lage ist alle dicht beieinander hängenden Buchstaben sauber voneinander zu trennen ohne dabei einzelne Buchstaben zu stark zu verändern. Eine stärkere Erosion sorgt dafür, dass einige Buchstaben, wie z. B. ein „R“ oft als „P“ erkannt werden und die Buchstaben oft nur Bruchstückhaft erkannt werden. Hier kann in einer späteren Optimierung mal ein Ansatz über mit einem Weichzeichner probiert werden.

Ein anderes Problem stellt Rauschen im Captcha dar. Bis zu einem SNR von etwa 5 dB besitzt der jetzige Ansatz eine Erkennungsrate von etwa 60% und ist damit relativ zuverlässig. Bei einem SNR von 0.01 dB sinkt die Erkennungsrate auf nur noch 10,6%. Das Rauschen soll eine Bildverarbeitung deutlich erschweren, was es definitiv auch tut. Captchas, in denen Linien und sonstige zusätzliche Symbole verwendet werden (wie z. B. in Abbildung 27 zu sehen), wurden in diesem Projekt außen vorgelassen, weil solche Captchas noch wesentlich schwerer zu verarbeiten sind. Da die Linien die Buchstaben miteinander verbinden, kann die Vorverarbeitung diese nicht mehr so gut trennen. Hier kann aber ein Ansatz über eine manuelle Selektion der Buchstaben durch den Nutzer und einer anschließenden Nachbearbeitung Abhilfe schaffen.

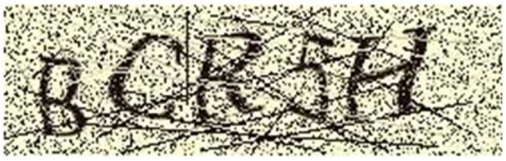

Abbildung 27: Captcha mit Linien und Rauschen

Zusammenfassen lässt sich sagen, dass als Ursache für viele Probleme die Vorverarbeitung der Captchas vermutet wird. Wenn diese Vorverarbeitung weiter optimiert wird, können Captchas besser erkannt und analysiert werden und ggf. lassen sich so auch die anderen Probleme wie Rauschen oder Störzeichen beheben. Die Erosion ist für stark verrauschten Bildern nicht optimal. Hier kann ggf. ein Ansatz mit anderen morphologischen Transformationen wie ein *Opening* zur Rauschreduktion probiert werden.

Anhang A: Aufbau des verwendeten LeNet

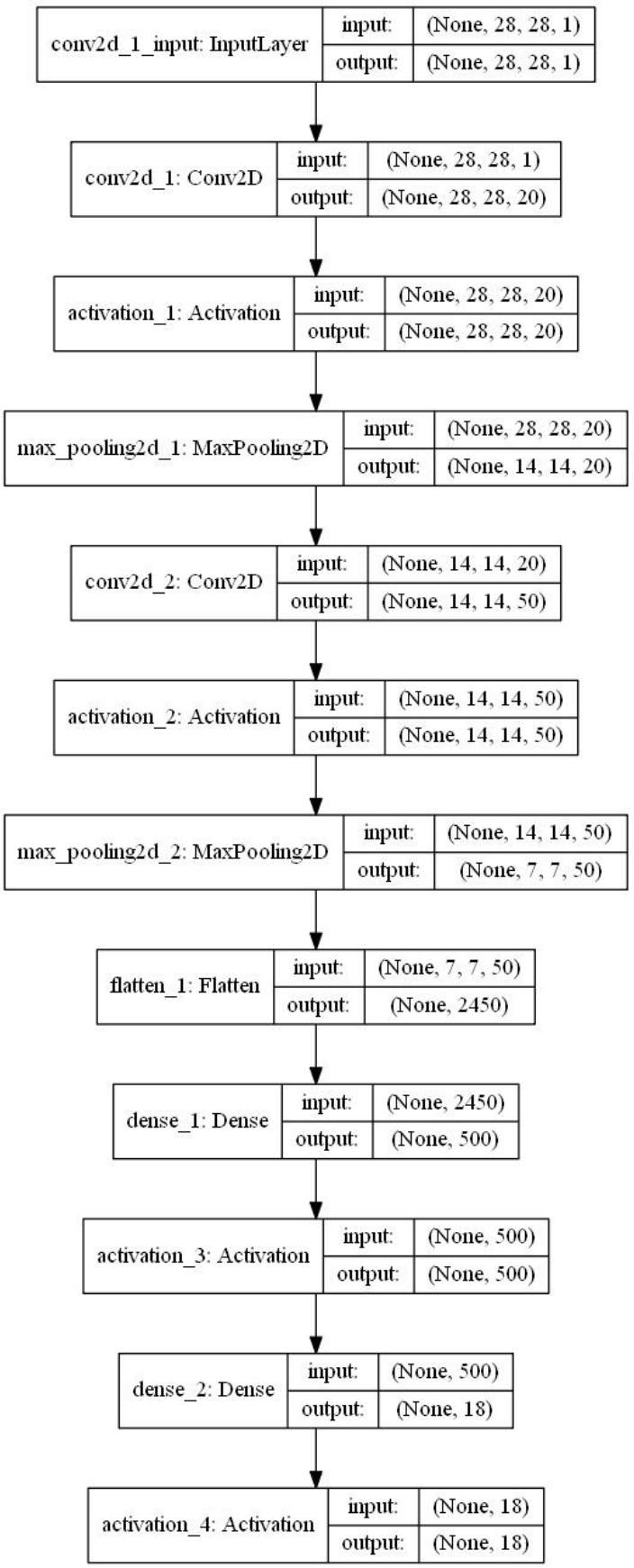

Abbildung 28: Baumstruktur des erzeugten Netzes

8 Literaturverzeichnis

[1] RajeshM, „Kaggle.com,“ 03 12 2017. [Online]. Available: https://www.kaggle.com/codingnirvana/captcha-images.

[2] Y. LeCun, L. Bottou, Y. Bengio und P. Haffner, „Gradient-Based Learning Applied to Document Recognition,“ 1989.

[3] D. A. Rosebrock, Deep Learning for Computer Vision with Python, 2017.