KI-Praxis III

Handschrifterkennung

Teil 3

Bislang wurden im Rahmen dieser Beitragsreihe lediglich numerische Daten klassifiziert. Damit können bereits viele Anwendungen abgedeckt werden. Man kann aber noch einen wesentlichen Schritt weiter gehen. So ist die „Erkennung“, also die Klassifizierung von Bildern, eine Fähigkeit, die lange nur dem Menschen vorbehalten war. Mit fortgeschrittenen Methoden der Künstlichen Intelligenz (KI) rückt diese Fähigkeit aber auch für Computer in den Bereich des Möglichen. In diesem Beitrag sollen zunächst Bilder von handgeschriebenen Ziffern im Fokus stehen. Auf diese Weise wird neben der Bilderkennung auch die Erfassung von Zeichen und Zahlen möglich. In späteren Beiträgen wird es dann allgemein um Bilderkennung und -klassifizierung gehen.

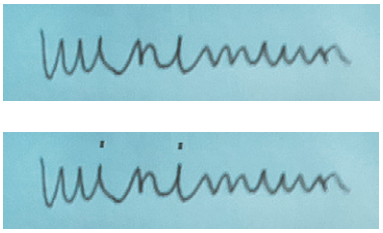

Bild 1: Bei Handschriften sind kleinste Details entscheidend.

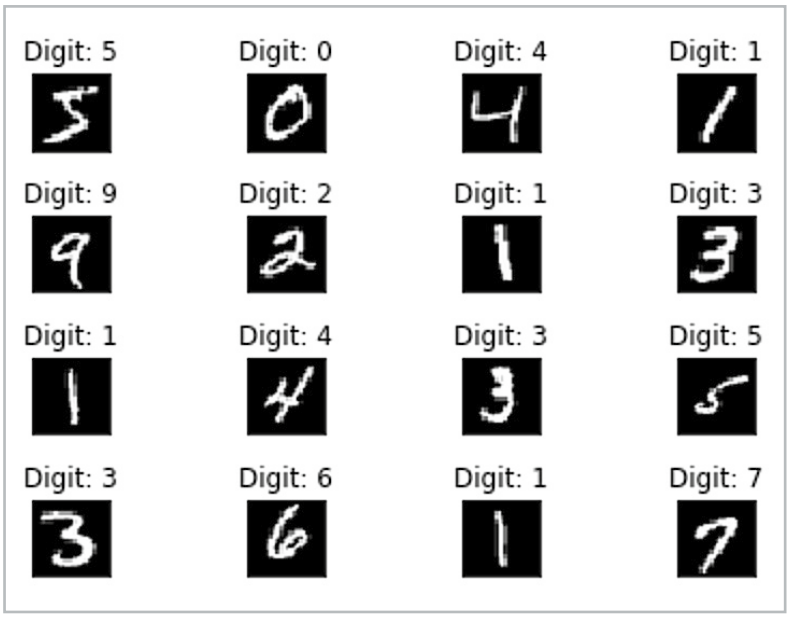

Bild 2: Ausschnitt aus dem MNIST-Datensatz nach [1]

Erkennung von handschriftlichen Zahlen

Die Handschrifterkennung hat zahlreiche Anwendungsgebiete wie beispielsweise die Schrifterkennung auf Formularen, Überweisungsbelegen und Adressen auf Briefen. Damit ist sie unter anderem für Banken, Behörden oder Postdienste von entscheidender Bedeutung.

Das Erkennen handgeschriebener Zahlen ist mit klassischen Programmiermethoden jedoch kaum realisierbar. Über viele Jahre hinweg wurde mit den verschiedensten Methoden wie Fourieranalyse oder Entscheidungsbäumen versucht, die menschliche Handschrift für Maschinen lesbar zu machen. Die Erfolge waren jedoch bestenfalls marginal.

Die enorme Vielfalt der Schrifttypen stellt die Hauptproblematik maschineller Handschrifterkennung dar. Bereits kleine Variationen können für die korrekte Erkennung eines Schriftzuges entscheidend sein. Bild 1 zeigt das handschriftliche Wort „Minimum“ einmal mit und einmal ohne i-Punkte. Während im ersten Fall praktisch nur eine Folge von geschwungenen Bögen erkennbar ist, wird das Wort im zweiten Fall von den meisten geübten Lesern problemlos erkannt.

Der Durchbruch kam auch hier erst mit der Anwendung künstlicher neuronaler Netzwerke. Diese sind weit besser als klassische Algorithmen dafür geeignet, die Aufgabe des maschinellen Handschriftenlesens zu lösen.

Für die Künstliche Intelligenz wurde das Erkennen von handgeschriebenen Zahlen so etwas wie das „Hello World“ in der Softwareentwicklung. Die Anwendung hatte ihren Durchbruch mit der zuverlässigen maschinellen Erkennung von Postleitzahlen bereits Ende der 1990er-Jahre. Der endgültige Erfolg kam, als Banken und Versicherungen die neue Technik zum automatischen Lesen von Formularen oder Überweisungsträgern übernahmen.

Der MNIST-Datensatz als Goldstandard

MNIST ist ein Akronym für „Modified National Institute of Standards and Technology“. Diese US-amerikanische Einrichtung stellt verschiedene Datensätze im wissenschaftlich-technischen Bereich zur Verfügung. In den letzten Jahren sind insbesondere auch umfangreiche Sätze für KI-Anwendungen hinzugekommen. Einer der bekanntesten Datensätze ist das „MNIST Handwritten Digit Classification Dataset“ [1]. Hierbei handelt es sich um einen Satz von 60.000 quadratischen Graustufenbildern mit jeweils 28 × 28 = 784 Bildpunkten (engl. „Pixel“ für Picture Elements) von handgeschriebenen Einzelziffern von 0 bis 9. Bild 2 zeigt einen Ausschnitt aus dem Datensatz.

Die Aufgabe für ein KI-System besteht darin, ein vorgegebenes Bild einer handgeschriebenen Ziffer einer von zehn Klassen entsprechend den Ziffern 0 bis 9 zuzuordnen. Die besten Modelle erreichen eine Klassifizierungsgenauigkeit von über 99,5 % bzw. Fehlerraten zwischen 0,5 % und 0,2 %. Damit werden sogar geübte menschliche Datentypisten übertroffen.

In diesem Beitrag sollen die Ziffern des MNIST-Datensatzes mithilfe eines neuronalen Netzes auf einem Raspberry Pi 4 klassifiziert werden.

Hierfür müssen die folgenden vier Libraries installiert sein und importiert werden:

```
import numpy as npy
import scipy.special
import matplotlib.pyplot as plt
import matplotlib
%matplotlib inline
import time   # for runtime measurements
```

Eine Kurzanleitung dazu findet sich im Download-Paket zu diesem Beitrag [2]. Die Details dazu wurden bereits in den letzten beiden Beiträgen [3] zu dieser Artikelserie dargelegt, sodass dort bei Bedarf nachgeschlagen werden kann. Der Aufbau des KI-Systems erfolgt wie in den letzten Beiträgen wieder in einem Jupyter Notebook.

Daten-Vorverarbeitung

Der MNIST-Datensatz kann von mehreren Quellen im CSV-Format (comma separated values) unter anderem von [4] aus dem Internet geladen werden. Man erhält einen Trainingssatz:

mnist_train.csv

mit 60.000 handschriftlichen Zahlen inklusive deren zugehörigen Nominalwerten. Dazu kommt ein Testdatensatz

mnist_test.csv

mit 10.000 Einträgen. Die Trainingsdaten können über

```
training _ data _ file = open(".../DATA/MNIST/mnist _ train.csv", r')            # 60000 entries
# training _ data _ file = open("...C:/DATA/MNIST/mnist _ train _ 100.csv", 'r')     # subset of 100 entries
training _ data _ list = training _ data _ file.readlines()
training _ data _ file.close()
print("number of training datasets loaded:", len(training _ data _ list))
```

in eine Python-lesbare Datenliste übernommen werden. Für

.../DATA/MNIST/mnist_train.csv

ist der Pfad, unter dem die csv-Dateien auf dem Raspberry Pi gespeichert sind, einzutragen. Nach der erfolgreichen Übernahme wird die Anzahl der Datensätze (z. B. 60.000) ausgegeben.

Es können entweder alle 60.000 Einträge der MNIST-Liste zum Training verwendet werden oder nur eine Auswahl von z. B. 100, 500 oder 1000 etc. Datensätzen. Im letzteren Fall muss die csv-Datei z. B. mit Excel oder LibreOffice etc. passend beschnitten werden. Die Daten sind zeilenorientiert, d. h., jede Zeile enthält einen kompletten Ziffern-Datensatz. Die gesamte Trainingsliste besteht also aus 60.000 Zeilen und 785 Spalten. Die erste Spalte gibt den Nominalwert („label“) der Ziffer an, die restlichen 784 Spalten enthalten die Graustufen für jedes Pixel der Zahlenbilder. Die überflüssigen Zeilen können einfach gelöscht werden.

Für die Bearbeitung sollte ein Rechner mit ausreichend Arbeitsspeicher zur Verfügung stehen, da die zu verarbeitenden Datenmengen vergleichsweise umfangreich sind. Empfohlen wird eine RAM-Größe von mindestens 4 Gigabyte.

Es ist zu beachten, dass die Trainingszeiten sehr lang werden können, wie die folgende Tabelle zeigt.

Anzahl Trainingsdaten	Raspberry Pi 4 mit 8 GB RAM	Rechner Dual Core @ 2,5 GHz, 4 GB RAM	Rechner Quad Core @ 3,5 GHz, 16 GB RAM
100	20 s	8 s	4 s
60.000	3 h	1,5 h	40 m

Dabei wurde ein Training mit zehn Epochen (Epoche = Durchgang durch alle Trainingsdaten) zugrunde gelegt.

Entsprechend können die Testdaten geladen werden. Auch hier ist der volle Datensatz mit 10.000 Einträgen bei Bedarf entsprechend reduzierbar.

Aufbau des neuronalen Netzes

Nachdem die Trainingsdaten in Python bzw. Jupyter zur Verfügung stehen, kann mit der Konstruktion eines neuronalen Netzes begonnen werden. Die Zahl der Eingangsknoten ergibt sich aus der Anzahl der Pixel pro Zahlenbild, also 28 x 28 = 784. Auch die Anzahl der Ausgabeknoten kann leicht festgelegt werden. Für eine sogenannte 1-aus-n-Kategorisierung (engl. „one-hot“) sind für zehn Ziffern auch exakt zehn Ausgabeknoten erforderlich. Jedem Knoten ist eine Ziffer von 0 bis 9 zugeordnet.

Wurde beispielsweise die Ziffer 3 erkannt liefert das Netz im Idealfall die Ausgabe:

Number	0	1	2	**3**	4	5	6	7	8	9
Probability	0	0	0	**1**	0	0	0	0	0	0

Ein Vorteil dieser Methode ist, dass man dabei auch ein gewisses Maß für die Zuverlässigkeit (Probability) der Ausgabe erhält. Ist sich das Netz „nicht ganz sicher“, ob es eine 5 oder eine 6 erkennt, könnte die Ausgabe z. B. so aussehen:

Number	0	1	2	3	4	**5**	**6**	7	8	9
Probability	0	0	0	0	0	**0,7**	**0,3**	0	0	0

Bei der Ausgabe über lediglich einen einzigen Knoten mit einem Wertebereich von 0...9 wäre eine solche Zusatzinformation nicht vorhanden.

Für eine erste Kategorisierung soll ein Netz mit drei Schichten zum Einsatz kommen. Neben der Ein- und Ausgabeschicht ist damit noch eine Zwischenschicht („hidden layer“) erforderlich.

Wie bereits im letzten Beitrag erläutert, kann die Anzahl der Knoten in der Zwischenschicht („hidden nodes“) nicht exakt berechnet werden. Ein guter Startpunkt liegt in diesem Anwendungsfall bei etwa dem Mittelwert zwischen Eingabe- und Ausgabeanzahl der Knoten, also z. B. 300 oder 400. Damit ergibt sich für den Aufbau des Netzes:

input nodes = 784
hidden nodes = 300
output nodes = 10

Neben der Anzahl der Knoten in jeder Schicht muss auch wieder eine Lernrate (LR, Wertebereich: [0 ... 1]) angegeben werden. Die Lernrate steuert, wie stark das Modell bei jedem Trainingsdurchlauf verändert wird. Zu kleine Werte führen zu langen Trainingsprozessen. Ein zu großer Wert kann zur Folge haben, dass das optimale Trainingsergebnis nicht gefunden wird oder der gesamte Trainingsprozess instabil wird.

Eine mögliche Strategie ist, zunächst mit einer relativ großen Lernrate (z. B. LR = 0,6) zu beginnen. Läuft das Training stabil ab, kann man den LR-Wert reduzieren und prüfen, ob so ein besseres Trainingsergebnis erzielt wird.

Training

Damit ist das Netz bereit zum Training. Hierfür wird das Notebook

MNIST_neural_network_numpy_RasPi_0V3.ipynb

aus dem Ordner „numpy“ im Download-Paket in ein Jupyter Notebook geladen.

Nach dem Start der entsprechenden Trainingszelle in Jupyter sollte sich die folgende Ausgabe ergeben:

```
epoch # 0 completed - elapsed time: 1051.073s
epoch # 1 completed - elapsed time: 2095.899s
epoch # 2 completed - elapsed time: 3142.525s
epoch # 3 completed - elapsed time: 4188.034s
epoch # 4 completed - elapsed time: 5234.445s
epoch # 5 completed - elapsed time: 6281.832s
epoch # 6 completed - elapsed time: 7326.559s
epoch # 7 completed - elapsed time: 8374.280s
epoch # 8 completed - elapsed time: 9419.153s
epoch # 9 completed - elapsed time: 10464.601s
runtime: 10464.602s
```

Das Training mit dem vollen 60.000er-Datensatz nimmt bei zehn Epochen auf dem Raspberry Pi 4 mit 8 GB eine Zeit von 10.465 Sekunden – also fast drei Stunden – in Anspruch.

Testen und vorhersagen

Nach dem Abschluss der Trainingsphase kann die Qualität des Netzwerkes überprüft werden.

Auch dies ist mit dem Notebook

MNIST_neural_network_numpy_RasPi_0V3.ipynb

möglich. Zunächst werden hierfür die Testdaten geladen:

```
# test _ data _ file = open("/home/pi/DATA/MNIST/mnist _ test.csv", 'r')  # 10000 entries
test _ data _ file = open("/home/pi/DATA/MNIST/mnist _ test _ 10.csv", 'r') # 10 entries
test _ data _ list = test _ data _ file.readlines()
test _ data _ file.close()
print("number of test datasets loaded:", end =" ")
print(len(test _ data _ list))
```

Es empfiehlt sich, mit einem kleinen Test-Datensatz von zehn bis 100 Einträgen zu beginnen. Diese Teildatensätze können wieder aus dem vollen Testsatz mit 10.000 Einträgen ausgeschnitten werden.

Die Netzwerk-Performance ergibt sich als Anteil von korrekt erkannten Ziffern zum Gesamtumfang des Datensatzes. Für ein Training mit einem reduzierten Datensatz wird typischerweise eine

performance = 0.7

bei zehn Epochen erreicht. D. h., 30 % aller Zahlenbilder werden nicht korrekt erkannt. Unter Verwendung des vollen Datensatzes ergibt sich meist eine Performance von ca. 93 %. Um etwas anschaulichere Ergebnisse zu erhalten, kann man sich auch einzelne Resultate im Detail ansehen.

Für eine korrekt erkannte Zahl ergibt sich z. B. Bild 3. Die in der Abbildung links stehende Balkenanzeige liefert dabei die direkte Netzwerkausgabe. Je heller das Feld dargestellt wird, desto höher ist die Bewertung des Netzes für die betreffende Zahl. Hier ist also der Vier der höchste Wert zugeordnet. Jedoch hat auch die Sieben eine gewisse, wenn auch deutlich geringere Wahrscheinlichkeit.

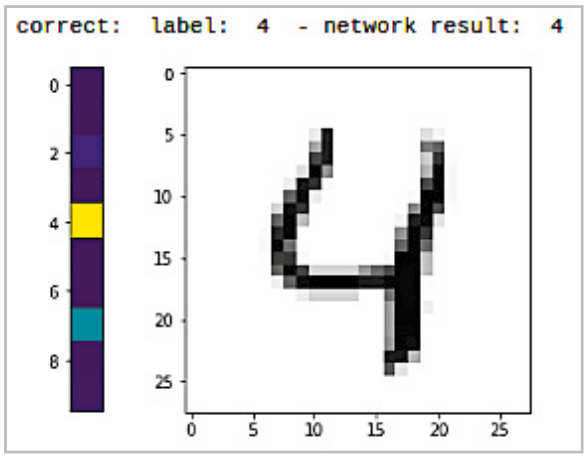

Bild 3: Korrekt erkannte Zahl

Interessanter sind die falsch erkannten Werte. Bild 4 zeigt beispielsweise eine als Neun erkannte Vier. Allerdings muss hier sicher eingeräumt werden, dass dieser Fehler auch einem menschlichen Betrachter unterlaufen könnte.

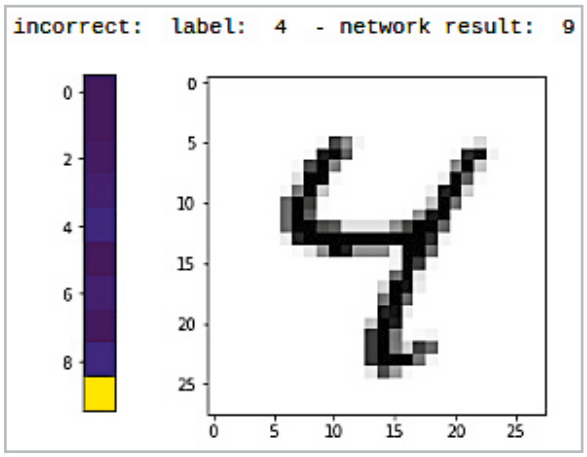

Bild 4: Falsch klassifizierte Ziffer

Im weiteren Verlauf des Datensatzes finden sich auch noch andere interessante Beispiele. So zeigt Bild 5 ein Beispiel, bei dem das Neuronale Netz keine eindeutige Antwort findet.

Der Nominalwert zu diesem Ziffernbild lautet Sieben. Das Neuronale Netz ist sich jedoch „unschlüssig“, ob es sich um eine Drei oder eine Neun handelt. Aber auch hier erkennt man, dass ein menschlicher Betrachter die Entscheidung des Netzes sicher nachvollziehen kann.

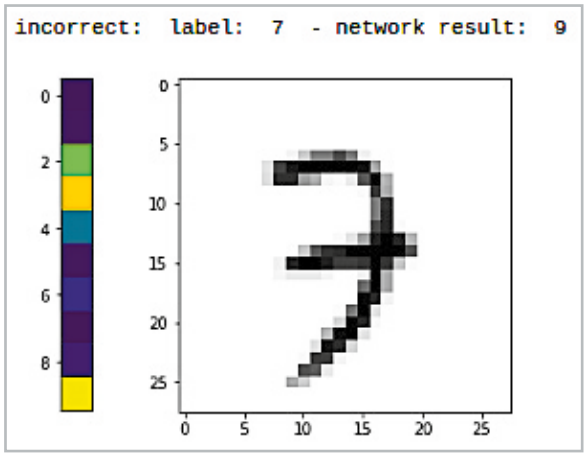

Bild 5: Drei, Sieben oder Neun?

Live-Erkennung mit PiCam

Das Erkennen von handgeschriebenen Ziffern eines vorgegebenen Datensatzes ist zweifellos ein wichtiger erster Schritt im maschinellen Lesen von Handschriften. Da allerdings nur wieder Daten aus dem gleichen Datensatz für den Test verwendet wurden, kann man das Vorgehen nicht als wirklich universell ansehen. Besser wäre es, wenn man beliebige Ziffern erkennen könnte, z. B. auch die eigene Handschrift. Auch das ist möglich.

Hierzu muss man zunächst eigene handgeschriebene Ziffern digitalisieren. Dies kann mit einem Scanner

aber auch über die Fotofunktion einer Digital- oder Handy-Kamera erfolgen. Nach der Aufnahme der Bilder müssen diese in das .png-Format mit einer Bildauflösung von 28 x 28 Pixel konvertiert werden. Das kann problemlos mit einem Bildbearbeitungsprogramm wie etwa IrfanView erfolgen.

Alternativ kann man die Raspberry Pi-Kamera (PiCam) und das Programm „ImageMagick“ auf dem Raspberry Pi verwenden.

Hinweis: Für erste Tests ist im Download-Paket ein Satz von in Eigenregie angefertigten Zahlen enthalten.

Die klassische PiCam verfügt über die folgenden Leistungsmerkmale:

Abmessungen:	25 x 20 x 9 mm
Sensor:	5 Megapixel mit Fixfokusobjektiv
Fotoauflösung:	bis 2592 x 1944 Pixel
Videoauflösung:	1920 x 1080 bei 30 Frames/s
	1280 x 720 bei 60 Frames/s
	640 x 480 bei 60 bis 90 Frames/s
Anschluss:	CSI via Flachbandkabel

Der Anschluss der Kamera erfolgt über die 15-polige serielle MIPI-Schnittstelle (CSI = Camera Serial Interface) auf dem Raspberry Pi. Der Vorteil dieser Schnittstelle gegenüber USB besteht in der direkten Verbindung von Kameramodul und dem Broadcom-Chip. Damit können auch bei höheren Auflösungen noch gute Bildwiederholraten erzielt werden.

Die CSI-Schnittstelle befindet sich zwischen der HDMI- und der Ethernet-Buchse. Um das 15-polige Flachbandkabel vom Kameramodul mit dem Board zu verbinden, zieht man den oberen Teil des CSI-Steckverbinders etwas nach oben, steckt dann das Flachbandkabel mit der blauen Markierung zum Ethernet-Anschluss hinein und drückt den Verschluss wieder nach unten. Damit ist der Kontakt hergestellt und das Kabel sitzt fest.

Nun muss noch der Kamera-Support in Raspbian aktiviert werden. Diese Aufgabe wird über das Konfigurationstool raspi-config erledigt. Dort die Kamera auf Enable setzen. Zum Abschluss ist noch ein Reboot erforderlich, damit die Kamera genutzt werden kann.

Das KameramModul wird über die beiden Programme raspistill (für Bilder) und raspivid (für Videos) angesprochen, die über zahlreiche Optionen verfügen. Auch eine Python-Bibliothek ist verfügbar.

Über die Anweisung

```
raspistill -o image.jpg
```

kann ein Testbild aufgenommen werden. Das Bild wird im Home-Verzeichnis (.../home/pi) abgespeichert. Wenn man nun die PiCam erfolgreich im Einsatz hat, kann man noch einen Schritt weiter gehen und die Live-Erkennung von handgeschriebenen Ziffern umsetzen. Das Ziel ist dabei, dass der Raspberry Pi eine Ziffer erkennt, sobald sie von der angeschlossenen PiCam erfasst wird.

Mit dem Raspberry Pi eigene Handschriften lesen

Die selbst erstellten Bilder können dann anstelle der MNIST-Daten analysiert werden. Bild 6 zeigt beispielsweise die sichere Erkennung einer Drei: Nur der Wert Drei zeigt eine helle Farbe in der Säule, alle anderen Werte sind dunkel.

Bei Bild 7 war sich die KI wieder „nicht ganz sicher“. Neben dem korrekten Wert Fünf hat auch die Sechs eine gewisse Wahrscheinlichkeit (grüne Einfärbung). Dennoch wurde das korrekte Ergebnis ausgegeben.

Aber nicht nur die Verarbeitung von vorgefertigten Bildern ist möglich. Vielmehr kann man mit der PiCam auch Live-Bilder erkennen. Hierzu wird das Programm

```
MNIST_numpy__PiCam_live_1V4.py
```

aus dem Downloadpaket auf dem Raspberry gestartet. Nach einer Trainingsphase erscheint ein Livebild der Kamera. Dieses zeigt das vorverarbeitete Gesichtsfeld der PiCam. Wird die Kamera auf eine Ziffer ausgerichtet, erscheint in der Shell der vom neuronalen Netz erkannte Zahlenwert. Bild 8 zeigt ein Beispiel für die Ziffer Vier.

Damit steht nun bereits ein Live-Analysesystem zur Verfügung, das z. B. auf

- die Auswertung von Autokennzeichen
- das Ablesen von Zählerständen oder
- die Erfassung von Hausnummern

erweitert werden könnte.

Training mit KERAS

Bislang wurde das Neuronale Netz nur unter Verwendung der allgemeinen numerischen Bibliothek „numpy“ aufgebaut. Seine wahre Stärke entfaltet Python

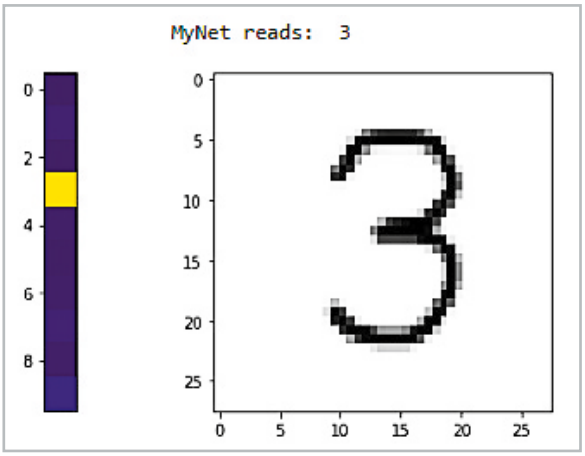

Bild 6: Sichere Erkennung einer Drei

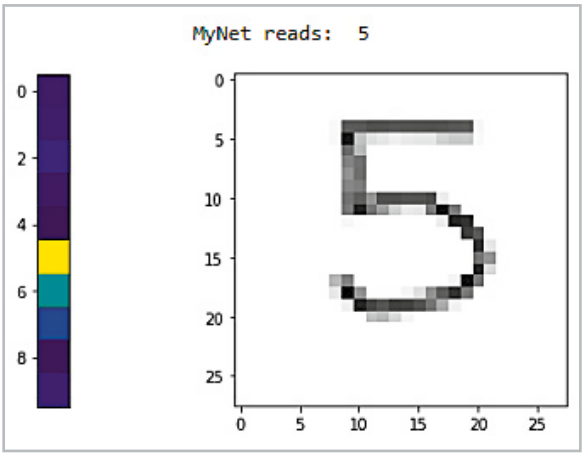

Bild 7: Hier ist sich das Neuronale Netz nicht ganz sicher.

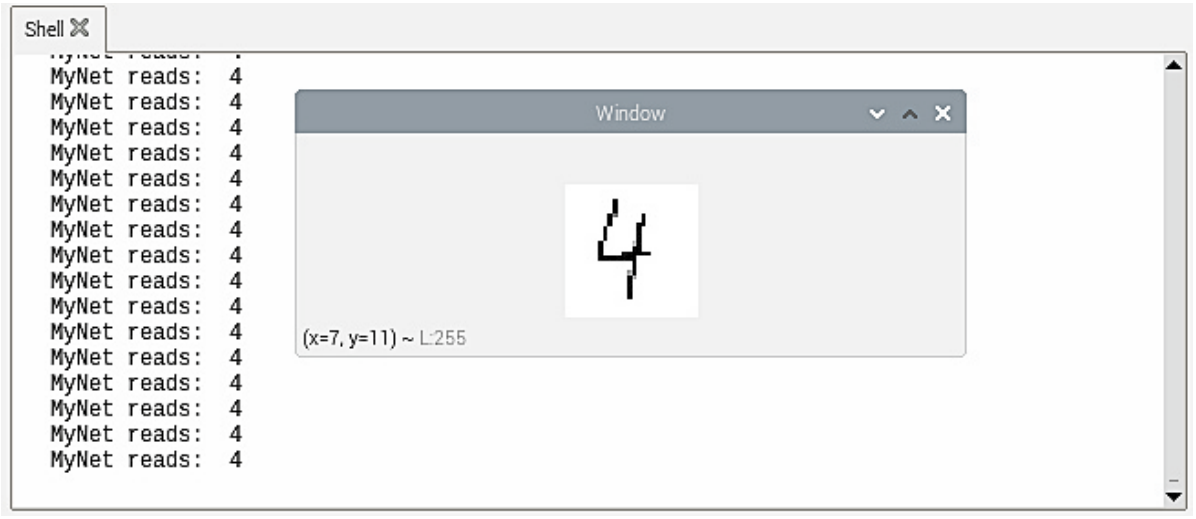

Bild 8: Der Raspberry Pi erkennt eine Vier im Live-Stream.

aber erst, wenn speziell auf Maschinenlernen ausgerichtete Libraries zum Einsatz kommen. Hier ist jedoch ein kurzer Hinweis angebracht:

Die Arbeit mit KERAS und Tensorflow auf dem Raspberry Pi erfordert tiefergehende Kenntnisse im Umgang mit Raspbian, Python und Jupyter etc. und ist daher für Einsteiger weniger geeignet!

Dafür steht mit KERAS eine umfangreiche Bibliothek für fortgeschrittene neuronale Netzwerk-Anwendungen zur Verfügung. Diese zeichnet sich durch schnelles Prototyping, ausgezeichnete Modularität und gute Erweiterbarkeit aus. Über KERAS können auch Deep-Learning-Frameworks wie Tensorflow, Theano oder CNTK in Python eingebunden werden. Dem effizienten Aufbau und Training eines neuronalen Netzwerks steht damit nichts mehr im Wege.

Zudem verfügt KERAS über eine einfache Möglichkeit, die Daten eines trainierten Netzes zu exportieren und zu speichern. Damit kann ein komplexes Netzwerk auf einem leistungsfähigen Rechner, eventuell mit GPU (Graphics Processing Unit, s. u.), trainiert werden. Anschließend werden die Daten dann auf einen kostengünstigen Kleinrechner wie den Raspberry Pi übertragen und stehen dort für spezielle Anwendungen zur Verfügung.

Ein Nachteil von KERAS ist allerdings, dass die Installation auf dem Raspberry Pi mit erheblichem Aufwand verbunden ist. Will man KERAS nur kurz testen, so sollte man es zunächst auf einem performanten Windows- oder Linux-Rechner ausführen. Dort ist die Installation über Anaconda mit dem Setzen eines Häkchens erledigt. Die Installation auf dem Pi dagegen ist aktuell nur für erfahrene Anwender empfehlenswert. Hier bleibt zu hoffen, dass zukünftige Entwicklungen die Installation deutlich vereinfachen werden. Dennoch soll im Folgenden kurz beschrieben werden, wie die Installation erfolgen kann. Die folgenden Libraries sind erforderlich:

- Tensorflow 2.2
- KERAS
- OpenCV

Die Installation von Tensorflow ist vergleichsweise aufwendig, da über die einfache Anweisung

pip install tensorflow

aktuell nur die Version 1.14. geladen wird. Details zur Installation der Version 2.2. finden sich im Download-Paket, sodass die Anweisungen direkt in das Terminal kopiert werden können. KERAS selbst dagegen kann problemlos über

pip install KERAS

geladen werden. Gleiches gilt für OpenCV:

sudo pip install opencv-contrib-python

Abschließend wird die PiCamera mit Numpy-Optimierungen geladen:

pip install „picamera [array]“

Im Folgenden wird ein sogenanntes tiefes neuronales Faltungsnetzwerk („deep convolutional neural network“) trainiert, um die handgeschriebenen Ziffern zu erkennen. Die Daten aus der Lernphase werden dann verwendet, um wieder über die Pi-Kamera Ziffern einzulesen und zu erkennen. Über KERAS wird auf Tensorflow zurückgegriffen. Um das ohnehin bereits aufwendige Projekt nicht zu komplex werden zu lassen, wurde auf eine Objektlokalisierung verzichtet. Die Zahlenbilder müssen daher direkt vor das Kameraobjektiv gehalten werden, damit sie vom Netzwerk identifiziert werden können.

Faltungsnetzwerke

Faltungsnetze (Convolutional Networks oder kurz ConvNets) bestehen aus Neuronen mit einerseits trainierbaren Gewichten und andererseits bestimmten voreingestellten Werten. ConvNet-Architekturen gehen explizit davon aus, dass die Eingabedaten aus Bildern bestehen. Dadurch können bestimmte Eigenschaften fest in der Architektur verankert werden. So wird die Auswertung effizienter und die Anzahl der erforderlichen Parameter im Netzwerk kann erheblich reduziert werden.

Ein Convolutional Neural Network ist in der Lage, Eingaben in Form einer Datenmatrix zu verarbeiten. Dies ermöglicht es, als Matrix dargestellte Bilder (Breite x Höhe x Farbkanäle) als Input zu verwenden. Ein konventionelles Neuronales Netz z. B. in Form eines Multi-Layer-Perceptrons (MLP) benötigt dagegen

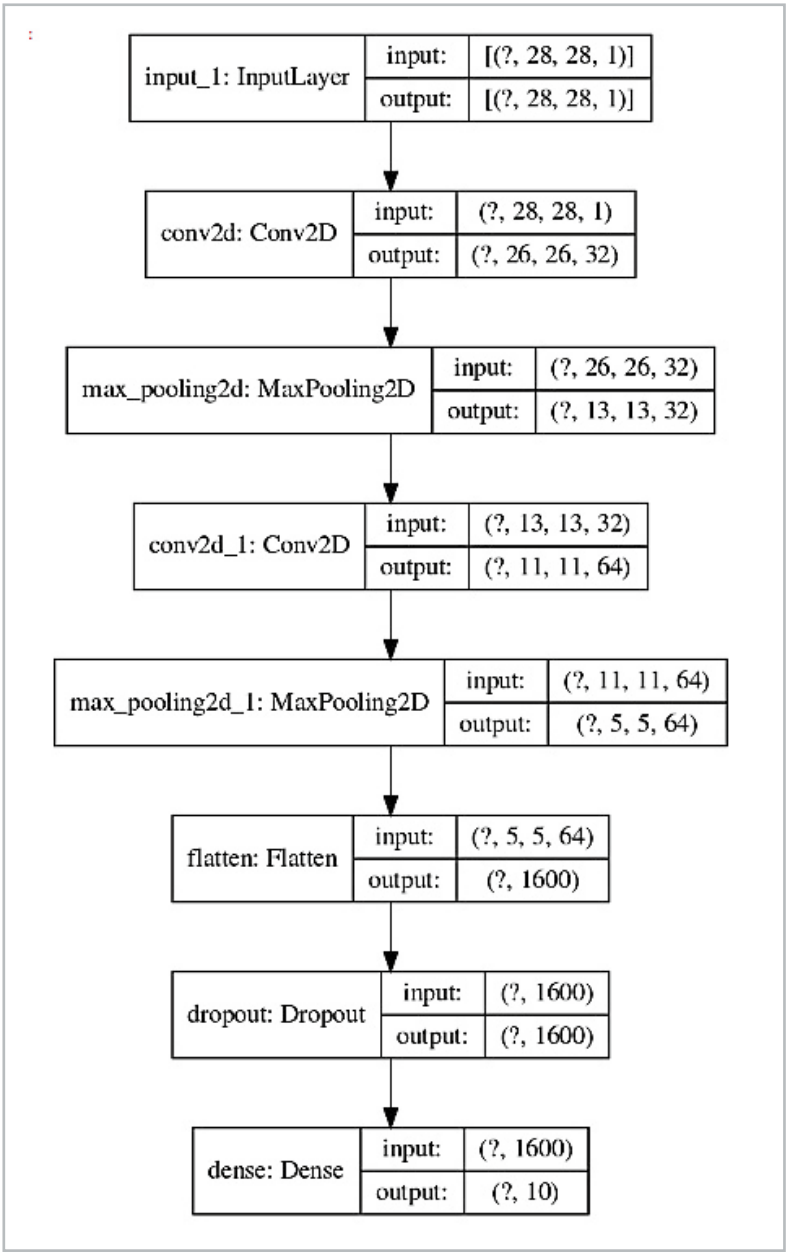

Bild 9: Aufbau des neuronalen Faltungsnetzwerks

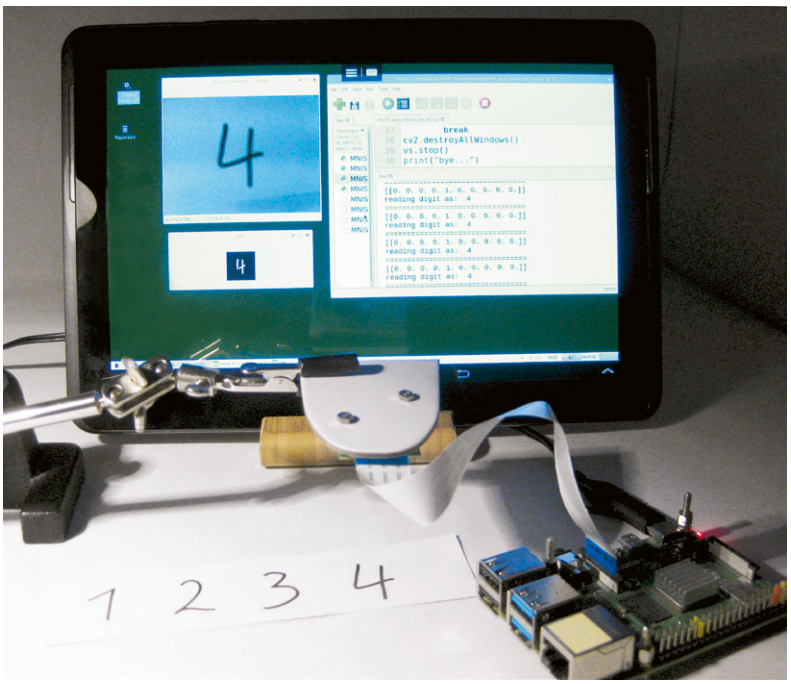

Bild 10: Live-Ziffernerkennung mit der PiCam

einen Vektor als Eingabeformat. Um ein Bild als Input zu verwenden, müssen also die Pixel des Bildes in einer langen Datenreihe vorliegen, so wie es im ersten Teil des Beitrags gezeigt wurde. Dadurch sind normale neuronale Netze z. B. nicht in der Lage, Objekte in einem Bild unabhängig von der Orientierung des Objekts im Bild zu erkennen. Das gleiche Objekt hätte beispielsweise nach einer leichten Drehung einen völlig anderen Input-Vektor.

Ein Faltungsnetzwerk erkennt dagegen mit seinen Filtern Strukturen in den Input-Daten. Auf der ersten Ebene werden die Filter dabei von einfachen Geometrien wie Linien, Kanten oder Farbflächen etc. aktiviert. Die Art der Filter ist dabei nicht fest vorgegeben, sondern wird vom Netz selbstständig gelernt. In der nächsten Ebene werden Strukturen erkannt, die bereits aus komplexeren Elementen wie z. B. Kurven, Bögen, Kreisen und Ellipsen etc. bestehen.

Mit jeder Filterebene erhöht sich so der Abstraktions-Level des Netzes. Nach weiteren Optimierungen können dann sogar komplette Gegenstände wie Kartons oder Fahrzeuge und sogar Menschen oder Tiere identifiziert werden. Details dazu werden in späteren Beiträgen zu dieser Artikelserie erläutert. Welche Abstraktionen schließlich zur Aktivierung der hinteren Layer führen, ergibt sich aus den charakteristischen Merkmalen der vorgegebenen Klassen. Für die genauere Erfassung der Funktionsweise eines Faltungsnetzwerkes kann es daher sehr interessant sein, die Muster zu visualisieren, die jeweils auf verschiedenen Ebenen zur Aktivierung der Filter führen.

MNIST mit ConvNet

Auch für die Anwendung eines ConvNets wird wieder der MNIST-Datensatz als Trainingsbasis verwendet. Das zugehörige Programm

MNIST_keras_Train_1V0.py

ist wieder im Download-Paket enthalten. Die Netzwerktopologie kann durch die Anweisung

model.summary()

ausgegeben werden und z. B. durch Bild 9 beschrieben werden:

Model: „sequential“

Layer (type)	Output Shape	Param #
conv2d (Conv2D)	(None, 26, 26, 32)	320
max_pooling2d (MaxPooling2D)	(None, 13, 13, 32)	0
conv2d_1 (Conv2D)	(None, 11, 11, 64)	18496
max_pooling2d_1 (MaxPooling2)	(None, 5, 5, 64)	0
flatten (Flatten)	(None, 1600)	0
dropout (Dropout)	(None, 1600)	0
dense (Dense)	(None, 10)	16010

Total params: 34,826
Trainable params: 34,826
Non-trainable params: 0

Man erkennt, dass dieses Netz bereits über 34.826 trainierbare Parameter verfügt. Unter Verwendung eines speziellen Grafikmoduls [5] lässt sich das Netz auch grafisch darstellen (Bild 9).

In der grafischen Darstellung wird nochmals deutlich, dass das Netz Bilder mit 28 x 28 = 784 Pixel in Schwarz-Weiß (1 bit) als Eingangsdaten erwartet. Die

letzte Ebene dagegen ist eine vollständig verbundene Schicht („dense“), die den zehn Kategorien zugeordnet ist, die wiederum die zehn Ziffern (0 bis 9) darstellen.

Das „None“ (bzw. das „?“ in der Grafik) ist ein Platzhalter, der besagt, dass das Netzwerk mehr als eine Stichprobe gleichzeitig abarbeiten kann. Eine Eingabeform (28, 28, 1) würde bedeuten, dass das Netzwerk lediglich ein einziges Bild simultan verarbeiten könnte. KERAS-Netze beherrschen jedoch Stapelverarbeitung (siehe „batch-size“ in der Trainingsphase) in frei wählbarer Länge. Damit können mehrere Bilder gleichzeitig zum Training herangezogen werden. Hierbei ist zu beachten, dass große Stapel zwar zu kürzeren Trainingszeiten führen, jedoch auch größere Arbeitsspeicher erfordern.

Im Gegensatz zu den letzten Abschnitten können Training und Anwendung nun sehr einfach getrennt werden. Zuerst wird über das oben skizzierte Netzwerk das Erkennen von Ziffern auf einem leistungsfähigen Rechner (vorzugsweise mit GPU) trainiert. Dann werden die berechneten Gewichte des Netzwerks verwendet, um in einem Live-Kamera-Feed Ziffern zu erkennen. Dieser wird direkt von der Raspberry Pi-Kamera aufgenommen. Bild 10 zeigt den entsprechenden Aufbau dazu.

Trainieren des Netzwerks auf einem leistungsfähigen PC

Um das Netzwerk zu trainieren, muss die Python-Datei MNIST_keras_Train_1V0.py auf einem möglichst leistungsfähigen Rechner ausgeführt werden. Das Programm verwendet KERAS, um ein tiefes neuronales Netzwerkmodell zu definieren, zu kompilieren und nach Abschluss der Trainings- und Validierungsphasen die Gewichte des Netzwerks abzuspeichern.

Die folgende Tabelle gibt einen Überblick über die zu erwartenden Trainingszeiten:

Epochen	Raspberry Pi 4 8 GB RAM	Rechner Dual Core @ 2,5 GHz	Rechner Quad Core @ 3,5 GHz
1	600 s	200 s	80 s
12	2 h	40 h	15 min

Eine weitere Verkürzung der Zeiten auf wenige Minuten selbst für zwölf Epochen wäre beispielsweise durch den Einsatz einer NVIDIA-Grafikkarte und der Nutzung der GPU möglich. Um die GPU einzusetzen, müssen jedoch die zugehörige Version von Tensorflow und die ausführbare CUDA-Datei von NVIDIA installiert sein.

Nach Abschluss des Trainings speichert das Programm die berechneten Gewichte des Netzwerks als *.h5-Datei ab. Diese Datei wird auf den Raspberry Pi kopiert, und dient dort als Grundlage für die Ziffernerkennung im Live-Videostrom. Das Kopieren kann über USB-Stick oder Filezilla etc. erfolgen. Alternativ ist das Training auch auf dem Pi selbst ausführbar. In diesem Fall muss man allerdings mit Trainingszeiten von mehreren Stunden rechnen. Im Download-Paket sind auch einige .h5-Dateien als Beispiele enthalten. Diese können ebenfalls auf dem Raspberry verwendet werden.

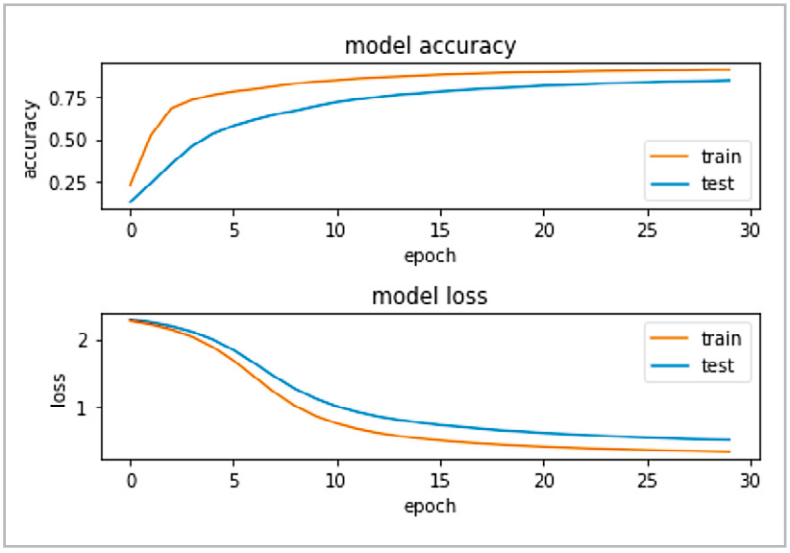

Bild 11: Accuracy- und Loss-Kurven beim Trainieren mit KERAS

Qualitätsbewertung

Bei der Bewertung der Qualität eines trainierten neuronalen Netzes spielen zwei Parameter eine wichtige Rolle:

- Loss-Funktion (Verlustfunktion)
- Accuracy-Wert (Genauigkeit)

Je geringer der „loss“-Wert (Verlust) eines Netzes, desto besser ist ein Modell trainiert. Der Verlust wird anhand von Training und Validierung berechnet und gibt an, wie gut das Modell für diese beiden Sätze abschneidet. Im Gegensatz zur „accuracy“ (Genauigkeit) ist der Verlust kein Prozentsatz. Es ist eine Zusammenfassung der Fehler, die für jedes Beispiel im Trainings- oder Validierungssatz gemacht wurden. Das Hauptziel in einem Lernmodell besteht üblicherweise darin, den Wert der Verlustfunktion über das Training hinweg zu reduzieren bzw. zu minimieren.

Beim Reduzieren des Verlustwerts gilt es jedoch, einige Feinheiten zu beachten. Beispielsweise kann es zum Problem der „Überanpassung“ kommen, bei dem sich das Modell die Trainingsbeispiele so genau „merkt“, dass beim Testsatz wieder mehr Fehler entstehen.

Eine Überanpassung tritt auch auf, wenn ein sehr komplexes Modell mit einer unverhältnismäßig hohen Anzahl von freien Parametern verwendet wird. Hier werden die Parameter dann so exakt an den Trainingssatz angepasst, dass sie für andere Werte wieder schlechtere Ergebnisse liefern.

Die Genauigkeit eines Modells wird bestimmt, nachdem die Modellparameter optimiert und festgelegt wurden und kein Lernen mehr stattfindet. Dann werden die Testproben dem Modell zugeführt und die Anzahl der Fehler wird ins Verhältnis zu allen Ergebnissen gesetzt. Werden also beispielsweise 1000 Ergebnisse ermittelt und das Modell klassifiziert 987 davon korrekt, dann beträgt die Genauigkeit des Modells 98,7 %.

In Bild 11 sind typische Lernkurven dargestellt. Es zeigt sich, dass die Accuracy in Laufe des Trainings zu-, die Loss-Funktion dagegen abnimmt. Weiterhin wird klar, dass die Kurven eine Sättigung erreichen, sodass immer längere Trainingszeiten schließlich keinen Vorteil mehr bringen.

Erkennen von Livebildern der Ziffern

Nach dem Abspeichern der *.h5-Datei kann man die Erkennung sowohl von handschriftlichen als auch von gedruckten Ziffern testen. Die Vorhersagegenauigkeit hängt auch stark von der Beleuchtung und dem Bildwinkel ab und natürlich auch davon, wie eindeutig die Ziffern tatsächlich geschrieben wurden. Danach kann das Programm

MNIST_keras_PiCam_live_1V0.py

gestartet werden. Nach dem Laden des Parametersatzes wird ein Live-Kamerabild dargestellt. Nun muss die zu identifizierende Ziffer in

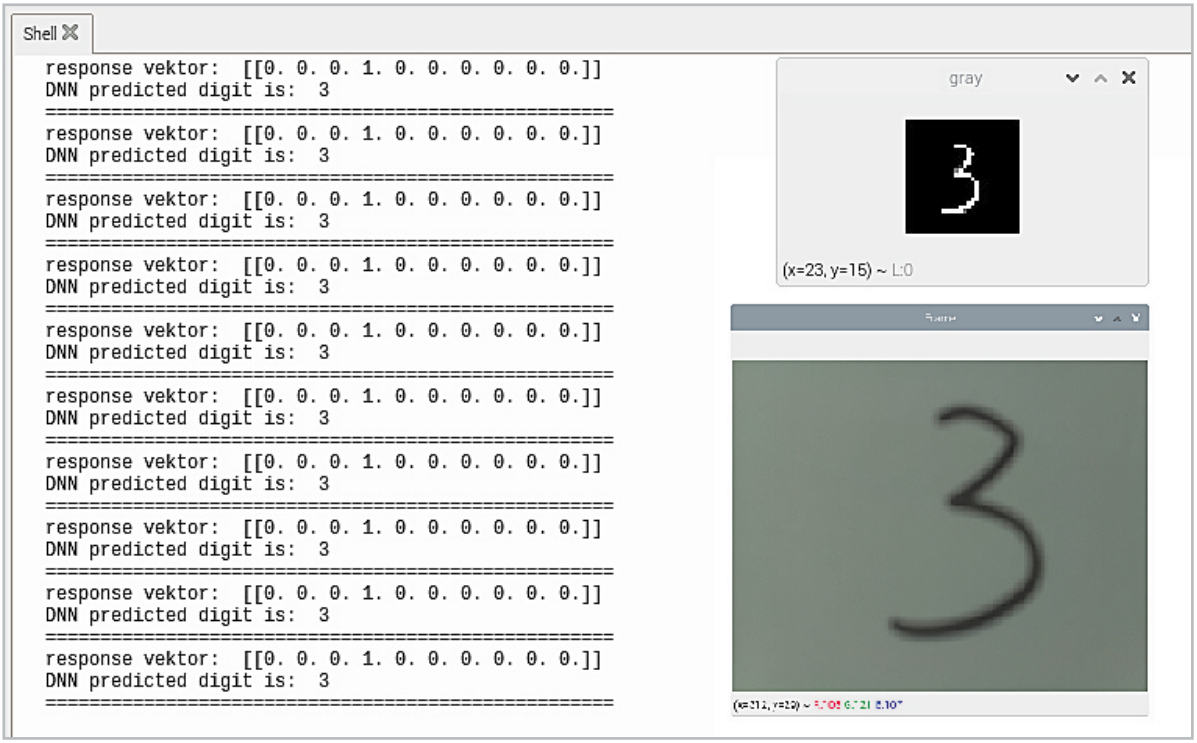

Bild 12: Die Ziffer Drei wurde korrekt erfasst.

das Blickfeld der Kamera gebracht werden. Dann wird über das Drücken der Taste „a“ (für „analysieren“) ein Schnappschuss des Kamerabildes aufgenommen. Anschließend wird automatisch eine Reihe von Transformationsschritten ausgeführt, bevor das Bild an das Neuronale Netz weitergeleitet wird. Zusätzlich zum ursprünglichen Videobild wird dabei auch eine vorverarbeitete Schwarz-Weiß-Version des Datenarrays ausgegeben. Diese kann als zusätzliche Kontrolle für die Bildqualität dienen.

Schließlich wird der vom Netzwerk errechnete Wert ausgegeben. Bild 12 zeigt ein Ergebnis für die Ziffer Drei.

Der Algorithmus im Detail

Das komplette Programm zur Erfassung von handgeschriebenen Ziffern sieht so aus:

```
#!/usr/bin/env python

print("importing libs...")
from skimage import img_as_ubyte
from skimage.color import rgb2gray
import cv2, imutils, time
from imutils.video import VideoStream
from keras.models import load_model

print("load model...")
model=load_model('MNIST_trained_model_RasPi401_30_epochs__001.h5')
model.summary()

vs=VideoStream(usePiCamera=["picamera"]).start()
time.sleep(1.0)

while True:
    frame=vs.read()
    frame=imutils.resize(frame, width=400)
    cv2.imshow("press 'a' to analyze - 'q' to quit", frame)
    key=cv2.waitKey(1)&0xFF
    if key == ord("a"):
        img_gray=rgb2gray(frame)
        img_gray_u8=img_as_ubyte(img_gray)
         (thresh, img_bw)=cv2.threshold(img_gray_u8,128,255,cv2.THRESH_BINARY|cv2.THRESH_OTSU)
        img_resized=cv2.resize(img_bw,(28,28))
        img_gray_invert=255-img_resized
        cv2.imshow("gray", img_gray_invert)
        img_final=img_gray_invert.reshape(1,28,28,1)
```

```
            ans = model.predict(img_final)
            print(ans)
            ans = ans[0].tolist().index(max(ans[0].tolist()))
            print('reading digit as: ',ans)
            print("================================")
        if key == ord("q"):
            break
cv2.destroyAllWindows()
vs.stop()
print("bye...")
```

Nach dem Importieren der erforderlichen Bibliotheken wird zunächst das vortrainierte Modell geladen:
model=load_model(‚MNIST_trained_model_RasPi401_30_epochs__001.h5‘)

Anschließend wird der Videostrom der PiCam gestartet. In der Hauptschleife wird das empfangene Videobild laufend aufgenommen und in einem Fenster dargestellt. Nach dem Drücken der Taste „a“ wird das Farbbild als Array von Gleitkommazahlen erfasst. Das RGB-Format wird dann in ein Graustufenbild konvertiert. Der nächste Schritt besteht darin, das Gleitkommaformat des Bildes in eine 8-Bit-Zahl mit einem Werte-Bereich von 0 bis 255 zu konvertieren.

Über OpenCV wird die Schwellenwertbildung durchgeführt. Die sogenannte „Otsu-Methode“ wird verwendet, um das Bild automatisch mit einem Schwellenwert zu versehen. Die Merkmale der Zahl werden damit wesentlich klarer erkennbar. Die nach Nobuyuki Otsu benannte Methode führt eine automatische Bildanalyse durch. Der Algorithmus gibt einen einzelnen Intensitätsschwellenwert zurück, der Pixel in zwei Klassen – Vordergrund und Hintergrund – unterteilt. Danach werden die Farben noch invertiert, da der zum Training verwendete MNIST-Satz die Zahlen in Weiß auf einem schwarzen Hintergrund enthält (s. Bild 2, 10 und 12) und die Bilder daher ebenfalls in dieser Darstellung vorliegen müssen.

Nach Abschluss dieser Vorbearbeitung kann das Bild schließlich an das vortrainierte ConvNet übergeben werden, das eine Vorhersage für die erfasste Ziffer liefert.

Das Ausgabearray repräsentiert wieder die Wahrscheinlichkeiten für die einzelnen Ziffern. Man erhält also ein Ausgabearray mit zehn Klassen, in dem für alle Ziffern von 0 bis 9 die Wahrscheinlichkeiten angegeben werden. Daraus wird schließlich der Wert mit der höchsten Wahrscheinlichkeit als Vorhersagewert ausgewählt und ausgegeben.

Fazit und Ausblick

In diesem Beitrag wurde dargelegt, wie man mithilfe eines kostengünstigen Kleinrechners neuronale Netze in der Praxis einsetzen kann. Zusammen mit der PiCam ist der Raspberry durchaus auch für Bildverarbeitungsaufgaben gerüstet. Neben einem einfachen neuronalen Netz kam dafür auch ein sogenanntes ConvNet zum Einsatz, dessen Eingangsschichten speziell für die Bildverarbeitung optimiert sind. Über die KERAS-Bibliothek können damit auch komplexere Netzwerke einfach realisiert werden.

Zudem konnten damit Training und Anwendung des Netzes problemlos getrennt werden. Einem raschen Training auf einem schnellen Rechner und der Anwendung des trainierten Models auf einem Kleinrechner steht damit nichts mehr im Wege.

In nächsten Beitrag wird es um die Erfassung und Erzeugung akustischer Signale gehen. Insbesondere die Spracherkennung und -erzeugung soll dabei im Fokus stehen. In den dann folgenden Artikeln wird dann wiederum unter anderem auf die Bildverarbeitung eingegangen. Dann werden auch fortgeschrittenere Methoden der Objekt- und Gesichtserkennung vorgestellt. ELV

Material	Artikel-Nr.
Raspberry Pi 4 Model B, 8 GB RAM	250567
Raspberry Pi Kameramodul v2	125073

Weitere Infos:

[1] MNIST-Datensatz: http://yann.lecun.com/exdb/mnist/
[2] Download-Paket zu diesem Beitrag: Artikel-Nr. 252233
[3] Serie „Künstliche Intelligenz“:
KI-Praxis I – Einstieg in die Künstliche Intelligenz, ELVjournal 3/2021: Artikel-Nr. 252090
KI-Praxis Teil II – Neuronale Netze – Aufbau und Training, ELVjournal 4/2021: Artikel-Nr. 252174
[4] MNIST-Datensatz Download als CSV-Datei: https://pjreddie.com/projects/mnist-in-csv/
[5] https://www.graphviz.org/

Alle Links finden Sie auch online unter: de.elv.com/elvjournal-links