KI für den ESP32

Auch auf Mikrocontrollern mit vergleichsweise wenig RAM und geringem CPU-Takt können neuronale Netze anspruchsvolle Aufgabe übernehmen. Wir zeigen in einer dreiteiligen Artikelstrecke Schritt für Schritt wie das funktioniert.

von Josef Müller

Kurzinfo

» **Bilder für Neuronale Netze vorbereiten**
» **Python-Skripte mit Jupyter-Notebooks nachvollziehen**
» **Mehr Trainingsdaten dank Data Augmentation**

Checkliste

Kosten:
0 Euro

Zeitaufwand:
2 Stunden

Software:
Python und Anaconda Navigator

Mehr zum Thema

» Josef Müller, ESP32CAM liest Wasseruhr, Make 2/21, S. 14
» Pina Merkert, Bilderstürmer, c't 16/18, S. 174
» Daniel Bachfeld, Einstieg in KI, Make 6/18, S. 36

Alles zum Artikel im Web unter
make-magazin.de/x665

Wenn man an praktische Anwendungen für neuronale Netze denkt, kommt einem schnell der Gedanke an Sprachverarbeitung in großen Rechenzentren wie bei *Amazon Alexa* oder *Google Echo* oder auch Bildverarbeitung für autonomes Fahren. Beides verbindet man mit komplexer Software und teurer Hardware, die nur mit viel Aufwand programmiert und betrieben werden können.

Das muss aber dank neuer und moderner Bibliotheken mittlerweile nicht immer so sein. In der Make 2/21 haben wir ein Beispiel vorgestellt, welches eine neuronale Anwendung auf einem günstigen Mikrocomputer realisiert, ganz ohne Cloud-Anbindung. Es wurde eine neuronale Bilderkennung auf einem ESP32 mit integrierter Kamera kombiniert. Damit haben wir einen analogen Wasserzähler quasi digitalisiert und sehr günstig an die Datenerfassung und Haussteuerung angebunden (1). Das Ganze ist für weniger als 30 Euro sehr kostengünstig realisierbar. Die Idee dahinter lässt sich mit ein wenig Fantasie auch auf andere Bereiche anwenden. Das Projekt fand sehr viel Resonanz, darunter auch viele Nachfragen, wie es im Detail funktioniert und wie man es auf eigene Ziffern und Zähler anpassen könnte.

Wir wollen deshalb in einer dreiteiligen Artikelreihe die wesentlichen Grundlagen hinter dem Projekt erläutern und Schritt für Schritt erklären, wie man selbst Bilderkennung mit neuronalen Netzen entwickelt und auf dem ESP32 zum Laufen bekommt. Am Ende der Artikelserie solltet ihr idealerweise ein selbst trainiertes neuronales Netz zur Bilderkennung mit wenigen Zeilen Code auf eurem ESP32 zum Laufen bringen. Darum herum könnt ihr dann hoffentlich die eigenen Ideen und Anwendung verwirklichen.

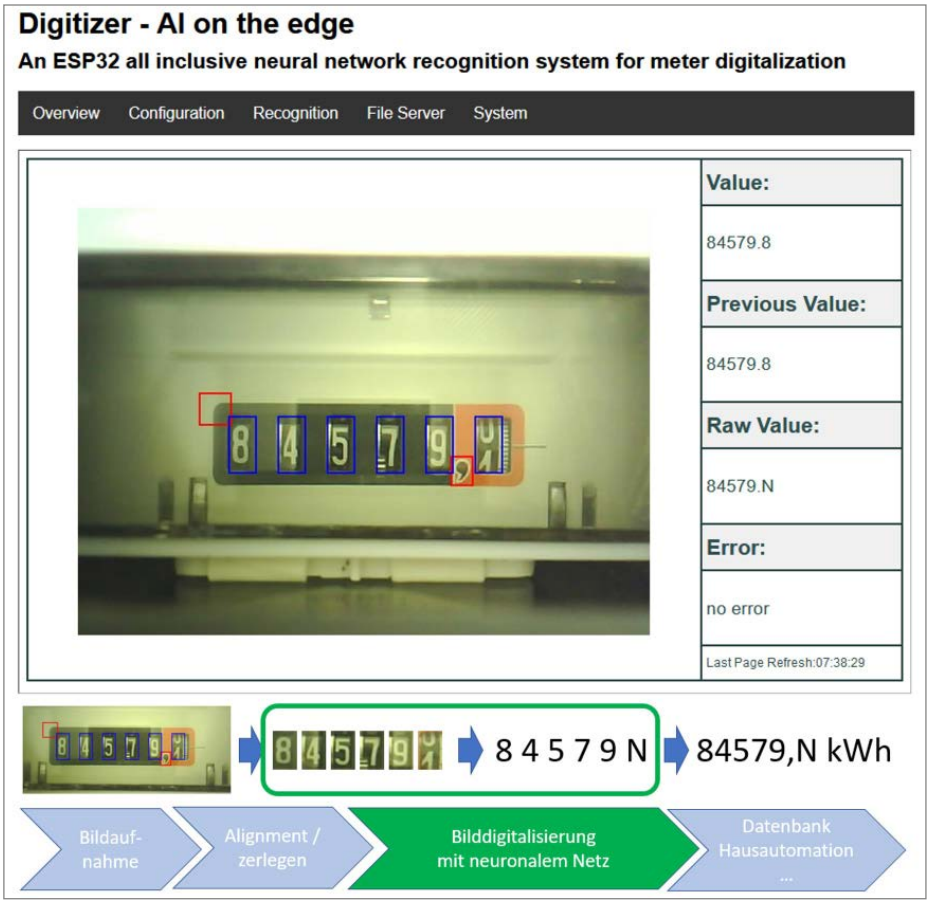
Alt und neu

Der große Vorteil von neuronalen Netzen in der Bilderkennung liegt darin, dass man recht einfach Objekte erkennen kann, ohne die Regeln dafür genau vorgeben zu müssen. Will man einem Menschen das Aussehen eines Dreiecks erklären, kann man ihm Dinge über Ecken, Seiten und Winkel und mögliche Konstellationen erzählen. Oder man zeigt ihm mehrere Bilder von Dreiecken und erklärt ihm, dies seien Dreiecke. In letzterem Fall werden die Regeln implizit im neuronalen Netz gelernt und hinterlegt.

Um dies zu verdeutlichen, nehmen wir ein komplexeres Beispiel: ein 20 × 30 Pixel großes Schwarzweißbild, welches eine Ziffer zwischen 0 und 9 zeigt. Dafür ein Regelwerk der Art „wenn dieses Pixel *weiß* und/oder diese *schwarz*´ ist, dann ...“ aufzustellen, um die richtige Ziffer zu erkennen, ist extrem aufwändig.

Selbst wenn die Zifferngröße und Position fix ist, sind dies doch recht viele und verschachtelte Regeln. Wenn dann die Ziffern auch noch an unterschiedlichen Positionen liegen und zu guter Letzt auch noch als Farbbild vorliegen, dann kann man sich leicht vorstellen, dass klassische Regelwerke mit „wenn … dann … sonst“-Konstruktionen sehr schnell überlastet sind.

In (2) sieht man unterschiedliche Ziffern 3, die alle durch ein neuronales Netz, welches im Wasserzähler verwendet wird, zuverlässig korrekt zugeordnet werden. Auch ein neuronales Netz kann man sich als sehr komplexes Regelwerk vorstellen. Auch hier gibt es Strukturen, die mit einem „wenn … dann …“-Konstrukt und einer Art Regelwerk vergleichbar sind.



(1) Der Zählerstand wird per neuronalem Netz digitalisiert.

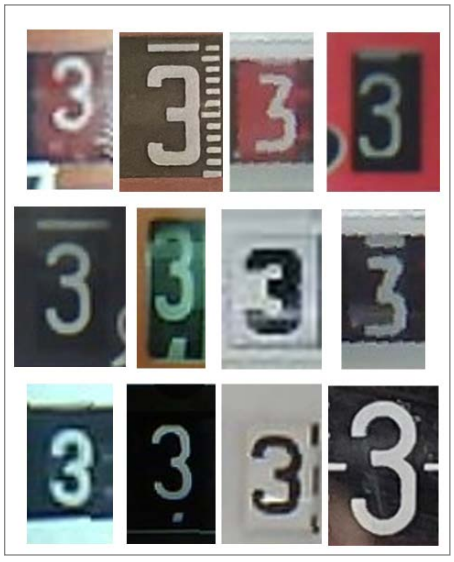
Artikelserie

Um sich dem Thema „KI auf ESP32“ schrittweise zu nähern, haben wir das Projekt zur Verwendung eines neuronalen Netzes auf drei Artikel aufgeteilt:

In **Teil 1** in diesem Heft geht es zunächst um die Beschreibung des Anschauungsbeispiels und hier im Schwerpunkt um die Trainingsdaten. Das Design und Training finden in einer Python-Umgebung auf einem normalen PC statt. Ziel ist es, Rohdaten in das notwendige Format zu konvertieren, die Eingangsbilder und Daten für das Training zu handhaben und das Prinzip der *Data Augmentation* zu verstehen. Dazu wird auf einem PC eine Python-Umgebung installiert und die grundlegenden Bibliotheken für die Bildvorbereitung und -konditionierung werden vorbereitet.

In **Teil 2** geht es um den Background, das Design und das Training des neuronalen Netzes. Auch dies findet aus Performance-Gründen noch in der Python-Umgebung des PCs statt. Das trainierte Netz wird für den Transfer auf den ESP32 vorbereitet. Am Ende dieses Teils ist man in der Lage, sein erstes selbst trainiertes neuronales Netz auf dem PC zu verwenden und eigene Bilder zu klassifizieren.

In **Teil 3** geht es dann konkret um ein neuronales Netz auf dem ESP32. Dort zeigen wir, wie man das mit Python erzeugte Modell eines neuronalen Netzes mit wenigen Zeilen C-Code auf den ESP32 portiert. Daraus lassen sich viele Ideen für anderen Netze ableiten, womit der Startschuss für eigene Projekte fällt.



2 Unterschiedliche Ziffernarten bei Wasser- und Stromzählern

3 Trainingsdaten für die Ziffern 0 bis 9 sowie nicht eindeutige Zwischenwerte (Not-a-Number)

Der wesentliche Unterschied zwischen klassischen Regelwerken und neuronalen Netzen liegt zum einen in der Struktur, wie die Informationen verarbeitet werden, und zum zweiten, wie die Regeln trainiert werden. Es werden nur Eingangsbilder und Angaben zu den erwarteten Ergebnissen zu diesen Bildern benötigt. Über einen Trainingsalgorithmus werden die Regeln sequenziell so angepasst, dass mit immer größerer Wahrscheinlichkeit das gewünschte Ergebnis herauskommt. Die Regeln entsprechen dann den Parametern des Netzes, etwa den sogenannten *Gewichten*. Weitere Details dazu erläutern wir dann im zweiten Teil.

Die Formulierung „mit immer größerer Wahrscheinlichkeit“ zeigt einen wesentlichen Knackpunkt dieser Art der Bildverarbeitung auf: Das Training der neuronalen Netze enthält einen zufälligen Anteil, sodass quasi kein trainiertes Netz identisch mit einem anderen ist, auch wenn dieselben Daten zum Training genommen werden. Zum zweiten kann man die internen Zustände im neuronalen Netz nicht klassischen Eigenschaften zuordnen, sodass man eigentlich nur anhand der Statistik der Endergebnisse eine Aussage über die Qualität machen kann.

ESP32 und TensorFlow

In den letzten Jahren gab es aber sowohl auf der Hardware- wie auch auf der Softwareseite wesentliche Weiterentwicklungen, womit es jetzt möglich ist, künstliche Intelligenz auch in DIY-Projekten mit günstiger Hardware und verhältnismäßig einfach anzuwendender Software zu verwenden. Auf der Hardwareseite gibt es den ESP32, der mit bis zu 240MHz Taktrate und 4MB PSRAM genügend Performance hat, um mittels neuronaler Netze Bilder zu erkennen.

Auf der Softwareseite ist mit der sehr mächtigen und frei verfügbaren Bibliothek *TensorFlow* von Google eine gut dokumentierte Implementierung für verschiedene Plattformen verfügbar. Seit einiger Zeit gibt es mit *TFlite* bzw. *tfmicro* eine dedizierte Ableitung für C-basierte Mikrocontroller-Entwicklungsumgebungen.

Der typische Wasser- oder Stromzähler besteht aus mehreren einzelnen Durchlaufrädern, die jeweils eine Ziffer des Zählerstandes darstellen. Das neuronale Netz soll den Zählerstand in digitaler Form zur Verfügung stellen. Um das neuronale Netz so einfach wie möglich und zugleich auch so flexibel wie möglich zu halten, soll es nur das Bild einer einzelnen Ziffer digitalisieren.

Das Thema Bildaufnahme und Zerlegung in die einzelnen Ziffern klammern wir hier mal aus. Wir setzen fertige Bilder der einzelnen Ziffern 0 bis 9 voraus. Da die Ziffern auf umlaufenden Ringen gedruckt sind, gibt es auch immer wieder unklare Zifferndarstellungen, wenn der Zähler zwischen zwei Zahlen steht. Dieser Status soll auch erkannt werden und mit einer speziellen 11. Klassifizierung, der sogenannten *Not-A-Number* (NaN) erkannt werden. 3 zeigt einen Satz typischer Beispielbilder und die Klasse NaN.

Vorbereitung

Das Ergebnis eines neuronalen Netzes ist immer nur so gut, wie die Eingangsdaten, die man zum Training zur Verfügung hat. Daher sollte man hier durchaus etwas Aufwand investieren. Idealerweise verwendet man zur Aufnahme der Bilder schon die gleichen Beleuchtungs- und Kameraeinstellungen, die später im finalen System Verwendung finden. Beim Wasser- bzw. Stromzähler wird mittels der OV2640-Kamera in der ESP32CAM die Oberfläche des Wasserzählers abfotografiert und dann in die einzelnen Ziffern zerlegt. Wichtig ist hier, darauf zu achten, dass die Orientierung, Position und Größe der Ziffer im einzelnen Bild vergleichbar sind. Als grobe Regel habe ich die Ziffern immer zentral mittig platziert und jeweils an der Seite einen Rand von ca. 15% gelassen (siehe auch 3).

Um später das neuronale Netz zu trainieren, muss man natürlich auch wissen, was auf dem einzelnen Bild zu sehen ist. Dieser Vorgang wird als *Labeling* (= Beschriftung) bezeichnet. Dabei wird jedem Bild der Inhalt zugeordnet, den man als Ausgabe des neuronalen Netzes erwartet. Dieser kann bei komplexeren Aufgaben auch aus vielen Informationen bestehen (Objekt, Farbe, Position, Objektdetails, …).

In umfangreichen Aufgaben kann das Labeling eine sehr aufwendige Aufgabe sein (z. B. Sprachdateien für Texterkennung). Hier ist es recht einfach: Man benötigt zu jedem Bild nur die Zahl, welche darauf zu sehen ist.

Die Verknüpfung von Bild und Inhalt kann auf verschiedene Arten gespeichert werden. Hier habe ich mich der Einfachheit halber entschieden, sie direkt an erster Position im Dateinamen zu speichern. So kann man später ohne Hilfsdatei etc. einfach den Zielwert direkt aus dem Dateinamen ablesen.

An dieser Stelle ein Vorgriff auf das Training: Wenn man feststellt, dass eine Ziffer immer hartnäckig falsch erkannt wird, lohnt es sich durchaus mal zu prüfen, ob sich beim Labeling des entsprechenden Bildes ein Fehler eingeschlichen hat. Wie schnell hat man sich dort vertippt und schon trainiert man das falsch beschriftete Bild einer 8 mit dem Zielwert einer 9.

Variationen

Idealerweise decken die Trainingsbilder des neuronalen Netzes alle möglichen vorkommenden Variationen der Eingangsbilder umfangreich ab. Das ist für die regulären Ziffern (0, 1, … 9) noch einfach machbar. Schwierig wird es schon bei der NaN-Klasse, denn man kann natürlich nicht von jedem möglichen Zwischenstand ein Bild erzeugen. Das ist sowohl was die Menge, als auch die Erzeugung der Bilder angeht, nicht möglich. Hier begnügt man sich mit typischen Beispielbildern. Daraus lernt das neuronale Netz dann die Abstrahierung auf die allgemeine Regel. Gleiches gilt auch für die anderen Eigenschaften eines Bildes, wie Helligkeit, Position der Ziffern, unterschiedliche Umgebungen etc.

Je mehr Trainingsdaten vorhanden sind, desto zuverlässiger erkennt das Netz später Ziffern. Nehmen wir mal an, es würde pro Ziffer nur ein einziges Bild zum Training verwendet. Und zufälligerweise wurde das Bild der Ziffer 0 bei etwas helleren Belichtungsverhältnissen aufgenommen (es war gerade das Kellerlicht an). Jetzt trainieren wir das Netz und alle Trainingsziffern werden hervorragend erkannt.

Voller Zuversicht verwenden wir das trainierte Netz in der Praxis. Plötzlich stellen wir aber fest, dass auch immer wieder andere Ziffern als 0 erkannt werden. Bei genauerer Analyse merken wir, dass dies immer dann der Fall ist, wenn das Kellerlicht gerade an ist. Unser neuronales Netz hat also gar nicht die Ziffer 0 gelernt, sondern erkennt eigentlich den Zustand „helleres Bild".

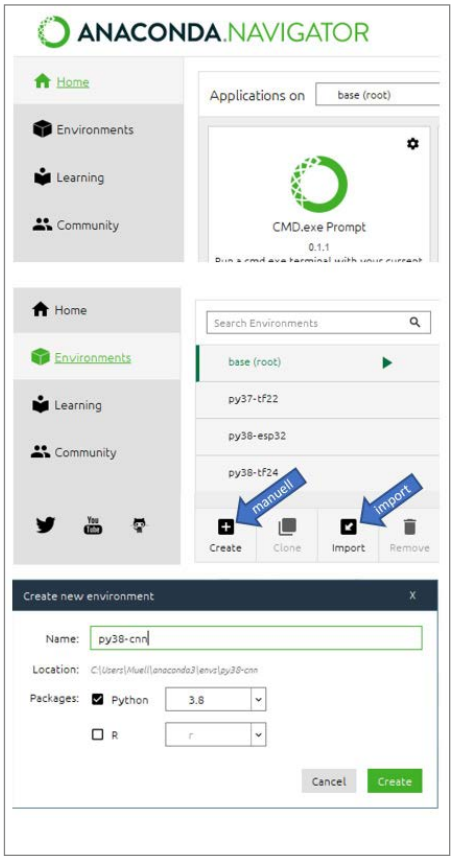
Da man nicht in das Innere des neuronalen Netzes reinschauen kann (zumindest nicht mit einfachen Mitteln), weiß man nicht, welche Eigenschaften der Bilder tatsächlich zur Klassifizierung führen. Daher ist es wichtig, möglichst viele dieser Variationen auch in den Trainingsdaten abzubilden. Je größer dabei das neuronale Netz ist, desto mehr Trainingsdaten werden auch benötigt.

Um nun nicht selbst hunderte Bilder für alle möglichen Varianten manuell aufnehmen zu müssen, gibt es die sehr mächtige Möglichkeit der *Data Augmentation*. Man vervielfältigt die Trainingsdaten, indem man typische Variationen künstlich erzeugt und in den Bildern z. B. Helligkeit und Rotation leicht verändert. Dies wird im letzten Abschnitt noch genauer erklärt, vorher richten wir aber noch die Trainingsumgebung ein, um die Trainingsbilder aufzubereiten.

Umgebung

Damit ihr euch nicht mit der teilweise doch recht aufwendigen Datenaufnahme beschäftigen müsst, haben wir für euch im GitHub-Repository einen ersten Datensatz an Rohdaten zur Verfügung gestellt. Dieser ist absichtlich nicht besonders groß und umfangreich, damit das Training auch im zweiten Teil nicht zu lange dauert und man verschiedene Parameter spielerisch variieren kann.

Für die Data Augmentation und zum Trainieren unseres Netzes nutzen wir eine Python-Umgebung mit *TensorFlow*. Unter Windows kann man Python sehr einfach mittels der



4 **Mit dem Anaconda Navigator kann man verschiedene Entwicklungsumgebungen installieren und verwalten.**

Plattform *Anaconda* installieren, verwenden und verwalten 4. Sie lässt sich selbst recht einfach installieren, man kann unterschiedliche Testumgebungen definieren und auch eine sogenannte *Jupyter*-Umgebung ist verfügbar. Mit Jupyter kann man im Browser quasi kommentierte Python-Skripte (*Notebooks* genannt) Schritt für Schritt ablaufen und sich Ergebnisse anzeigen lassen. Die genaue Installations- und Bedienungsanleitung und ein Video dazu findet ihr unter dem Link in der Kurzinfo.

```
from PIL import Image
image = Image.open('Size_Example_1.jpg')

size_x = 40
size_y = 64

for i in range(4):
    resize = image.resize((size_x, size_y), Image.NEAREST)
    print("Größe: " + str(size_x) + " x " + str(size_y) + " Pixel")
    display(resize)
    size_x = int(size_x / 2)
    size_y = int(size_y / 2)
```

```
Größe: 40 x 64 Pixel
Größe: 20 x 32 Pixel
Größe: 10 x 16 Pixel
Größe: 5 x 8 Pixel
```

5 **Skalierung des Testbildes**

```
import glob
import os
from PIL import Image

Input_dir = 'bilder_original'
Output_dir= 'bilder_resize'
target_size_x = 20
target_size_y = 32

#LÖSCHEN hier nicht abgedruckt

files = glob.glob(Input_dir + '/*.jpg')
for aktfile in files:
    print(aktfile)
    image = Image.open(aktfile)
    image = image.resize((target_size_x, target_size_y), Image.NEAREST)
    base=os.path.basename(aktfile)
    save_name = Output_dir + '/' + base
    image.save(save_name, "JPEG")
```

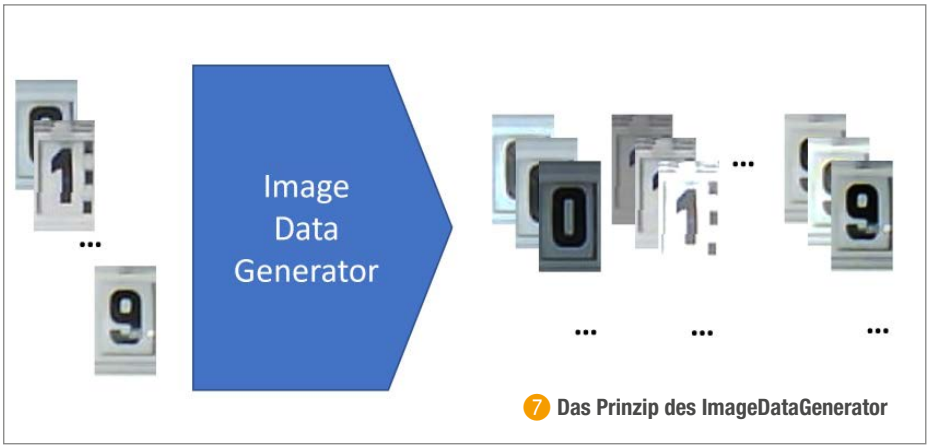
6 Skalierung aller Trainingsbilder auf eine einheitliche Größe

Trainingsbilder

Oft haben die Originaldaten unterschiedliche Größen, etwa weil der Ausschnitt aus den Rohdaten variiert oder einfach eine viel größere Auflösung bietet, als für die neuronale Berechnung notwendig ist. Für das später verwendete neuronale Netz muss eine einheitliche Eingangsgröße verwendet werden. Generell gilt, je kleiner die Bildgröße, desto schneller ist auch das spätere Training und die Erkennung. Andererseits gehen Informationen verloren, wenn die Bilder zu klein werden. Für meine neuronalen Netze hat sich folgende Daumenregel bewährt: Wenn *ich* auf dem verkleinerten Bild das Merkmal noch erkennen kann, dann ist es auch ausreichend für die automatische Erkennung durch ein neuronales Netz.

Das erste Python-Skript in Form eines Jupyter-Notebookes (*1_Test_Groesse_Trainingsbilder.ipynb*) lädt mittels der *Pillow*-Bildverarbeitungsbibliothek das Beispielbild und gibt immer kleiner skalierte Versionen aus 5. Zu beachten ist hier, dass als Skalierungsmethode keine Interpolation, sondern die Methode *Nearest* angewendet wird. Dies entspricht am ehesten der später auf dem ESP32 verwendeten Algorithmen. Man kann erkennen, dass in der kleinsten Version (5 × 8 Pixel) die Ziffer nicht mehr lesbar ist. Die Version 10 × 16 Pixel lässt sich noch sehr gut lesen und wäre somit gut geeignet. Wenn man an spätere Variationen mit deutlich dünneren Schriftarten denkt, so ist es sinnvoll, nicht die kleinste mögliche Größe zu nehmen, sondern mit der Größe 20 × 32 zu starten. Dies kann man mit dem Beispielbild 2 auch mal testen.

Nachdem die Größe nun auf 20 × 32 Pixel festgelegt ist, werden mit einem zweiten Skript alle Trainingsbilder auf die einheitliche Größe skaliert. Das ganze findet sich im Skript (*2_Skalierung_Trainingsdaten.ipynb*) 6. Zunächst werden neben der Bilderverarbeitungsbibliothek *Pillow* auch noch zwei Bibliotheken für die Dateibearbeitung geladen (`glob`, `os`). Im ersten Schritt wird das Zielverzeichnis (*Output_dir*) gelöscht und dann werden in einer zweiten Schleife alle Originaldateien geladen, skaliert und abgespeichert.



7 Das Prinzip des ImageDataGenerator

Data Augmentation

Praktischerweise bringt die Bibliothek von TensorFlow bereits Funktionen für Data Augmentation mit. Allgemein stellt man unter TensorFlow die Trainingsdaten als großes mehrdimensionales Array (Bilddaten und Zielwerten) bereit. Will man die Data Augmentation anwenden, muss man noch einen sogenannten *ImageDataGenerator* vorschalten. Diesen kann man sich wie einen Modulator vorstellen, der die Bilddaten als Eingangswerte bekommt, mit verschiedenen einstellbaren Eigenschaften modifiziert und dann wieder als Bild ausgibt 7.

Ausprobieren kann man das Beispiel der Helligkeitsvariation mit dem Skript *3_Augmentation_Helligkeit.ipynb* (siehe Link). Zunächst muss neben der Bilderverarbeitungsbibliothek noch die Funktion `DataIma-`

geGenerator aus der TensorFlow-Bibliothek geladen werden. Es wird wieder das bisherige Beispielbild verwendet. Das neuronale Netz und auch der Datengenerator erwarten ein Array von Trainingsdaten, also genau genommen ein 4-dimensionales Array, wobei der erste Index über die Anzahl der Trainingsbilder iteriert und die anderen die Bildinformationen enthalten (x, y, RGB). Zu Demonstrationszwecken zeigen wir hier nur ein einzelnes Trainingsbild, weshalb einige Operationen mit der Python-Bibliothek *numpy* notwendig sind, um es (mit den Funktionen `array` und `expand_dims`) in ein Format umzuwandeln, mit dem der `ImageDataGenerator` umgehen kann.

Der Generator lässt sich recht einfach nutzen, wie in den folgenden Zeilen zu sehen ist:

```
Brightness_Range = 0.5
datagen = ImageDataGenerator(
brightness_range = [1-Brightness_
Range,1+Brightness_Range])
iterator = datagen.flow(sample, batch_
size=1)
```

Zunächst muss man definieren, welcher Parameter in welchen Grenzen variiert werden soll. Der Parameter für die Helligkeit heißt `brightness_range` (die klein geschriebene Version im Listing) und erwartet ein Array mit der minimalen und maximalen Helligkeitsveränderung (1 = unverändert, 0 = dunkel, 2 = weiß). Hier wird eine Variation mit 50% (=0.5) verwendet. Jetzt muss man noch definieren, auf welche Bilddaten die Variation angewendet werden soll. Dies ist hier als Iterator definiert und besteht nur aus dem einzelnen Testbild `sample` (mit der `batch_size=1`) – zu letzterem im nächsten Teil dann mehr.

Jetzt bekommt man über die Funktion `iterator.next()` immer ein neues Bild, auf welches zufällig die Helligkeitsvariation angewendet wurde. Da es nur ein einziges Eingangsbild gibt, sieht man sehr schön, wie die Funktion arbeitet und aus quasi einem einzigen Bild eine beliebige Anzahl an Trainingsbildern erzeugt. Und damit ist auch klar, warum ein so trainiertes neuronales Netz jetzt eben nicht mehr den Zustand der Kellerlampe lernt, sondern unabhängig von der Bilderhelligkeit die darauf abgebildete Ziffer (8).

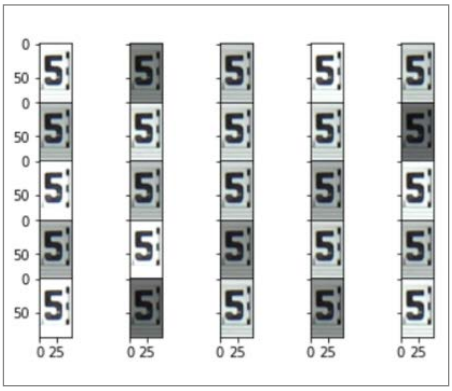
Neben der Bildhelligkeit gibt es noch eine ganze Menge anderer Parameter, die über diese Funktion variiert werden können (siehe Tabelle *Variationen für Bilder*).

Der Einfluss dieser Parameter ist im letzten Python-Skript *4_Augumentation_All.ipynb* anhand zunächst der Variation der einzelnen Parameter und dann auch anhand einer Kombination aller zu sehen. In diesem Beispiel kann man auch verschiedene Parameter durchtesten und auch eigene Bilder prüfen. Das letzte Beispiel mit der Variation aller Parameter zeigt auch, dass man sich genau überlegen muss, wie weit die Parameter gestreut werden sollen (9). Nicht bei allen Bildern kann man noch erkennen, dass es sich ursprünglich um die Ziffer 5 gehandelt hat. Daraus kann man ableiten, welche Variationen noch sinnvoll sind. Das ist eine gute Übung zur Vorbereitung für den nächsten Teil.

Ausblick

In diesem Teil wurde die Trainingsumgebung Anaconda eingeführt und auf die Vorbereitung der Trainingsdaten eingegangen. Ein wichtiger Aspekt ist die künstliche Variation der Trainingsdaten, um später ein möglichst robustes neuronales Netz zu erhalten.

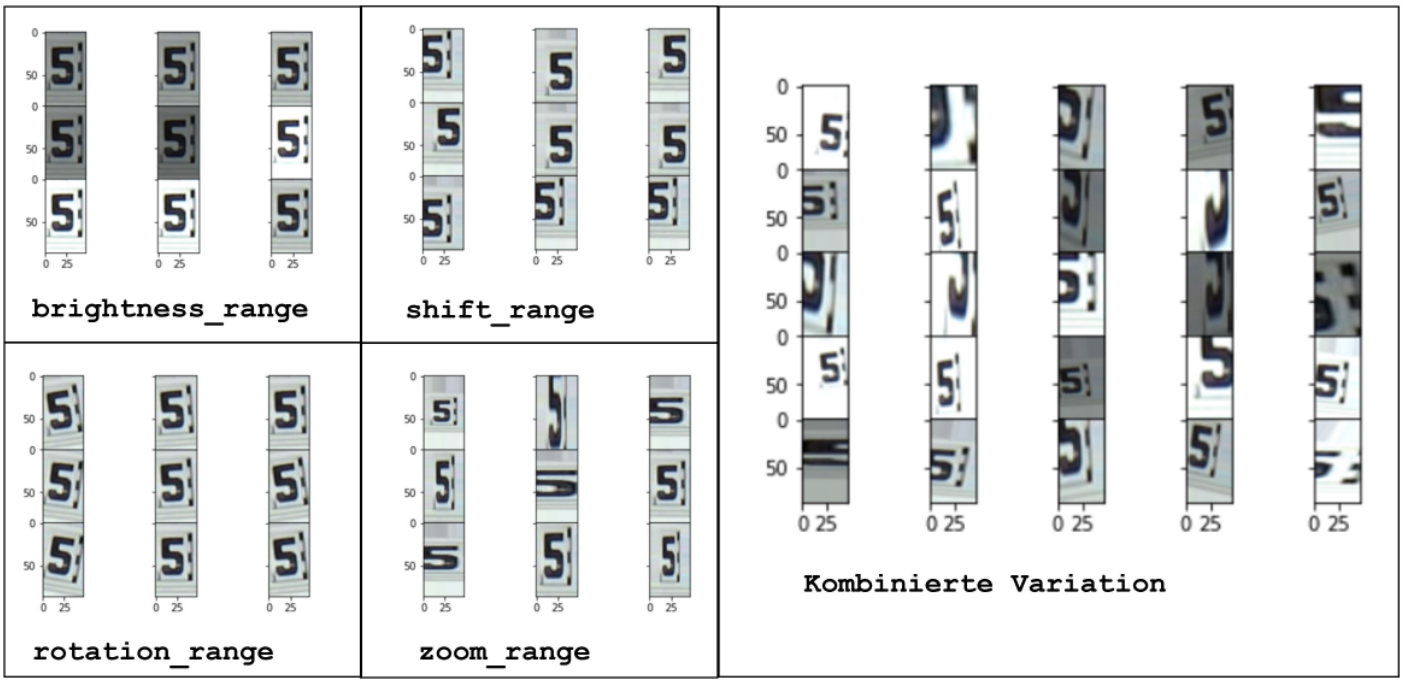
Viel Spaß mit den ersten Experimenten zu den Trainingsdaten und eventuell dem Aufnehmen und Erzeugen von eigenen Beispielen. Vielleicht kann das neuronale Netz im nächsten Teil dann schon mit den eigenen Problemen trainiert und damit geübt werden! *—dab*



(8) **Durch Data Augmentation werden aus einem Ursprungsbild viele weitere Bilder mit unterschiedlicher Helligkeiten generiert.**

Variationen für Bilder

Art	Parameter
Helligkeit	brightness_range
Verschiebung	width_shift_range height_shift_range
Verdrehung	rotation_range
Verzerrung	zoom_range



(9) **Weitere durch Data Augmentation generierte Bilder der Ziffer 5**