Teil 2

KI für den ESP32

Mit Python und Tensorflow modellieren und trainieren wir mit wenigen Zeilen Code interaktiv ein neuronales Netz, das Ziffern von Wasser-, Strom- und Gaszählern erkennen kann.

von Josef Müller

Im ersten Teil lag der Fokus auf der Installation der Python-Umgebung und vor allem der Vorbereitung der Trainingsdaten. Dabei wurde durch die Verwendung von *Data Augmentation* aus wenigen Trainingsbildern ein umfangreicher Datensatz generiert. Mit diesen Trainingsbildern soll nun im zweiten Teil ein neuronales Netz trainiert werden, das wir noch bauen müssen, was aber dank des grandiosen Frameworks *Tensorflow* gar nicht so schwer ist und auch wenig bis gar keine mathematischen Kenntnisse voraussetzt. Dennoch kurz ein wenig Theorie.

Der Begriff *neuronales Netzwerk* zeigt schon im Namen, an welche biologische Analogie diese Struktur angelehnt ist. Im Vergleich zwischen Nervenzellen und künstlichen neuronalen Netzen sieht man sehr ähnliche Strukturen: Nervenzellen/Neuronen, die als Eingang und Ausgang dienen, Verbindungen zwischen beiden und einen vielschichtigen Informationsfluss über ein komplexes Netzwerk mit vielen Verbindungen.

Die Idee hinter neuronalen Netzen ist es, eine Art selbstoptimierendes System zu schaffen. Neuronen sind die Knoten des Netzes. Man kann sich einen Knoten im Prinzip wie einen kleinen Rechner vorstellen (1), der mit seinen Eingangsdaten viele Multiplikationen und Additionen durchführt und das Ergebnis an seinem Ausgang weitergibt. In einem neuronalen Netz sind die Neuronen in Schichten bzw. *Layern* gruppiert, innerhalb derer sie jedoch nicht untereinander verbunden sind. Neuronen unterschiedlicher Schichten sind hingegen miteinander verbunden.

Übliche neuronale Netze haben eine Eingabeschicht für die Daten (hier unsere Bilder), mindestens eine (verborgene) Zwischenschicht (*Hidden Layer*) und eine Ausgabeschicht (zur Ausgabe der Antwort). Alle Neuronen des Hidden Layers sind typischerweise an ihren Eingängen mit allen Neuronen der vorherigen und am Ausgang mit allen Neuronen des folgenden Layers verbunden (2). Wenn jedes Neuron mit jedem Neuron des nächsten Layers verbunden ist, spricht man von *Fully Connected Layer* oder *Dense Layer*.

Die Eingänge eines Neurons werden für jede Verbindung mit einem individuellen Faktor multipliziert, der sogenannten *Gewichtung* (w, englisch *weight*). Es gibt so viele Gewichte, wie es Eingänge bzw. Verbindungen zu anderen Neuronen gibt. Im Training eines neuronalen Netzes werden die Werte für die Gewichte jeder einzelnen Verbindung in vielen aufeinander folgenden Durchgängen so lange angepasst, bis die Ausgabe näherungsweise zum erwarteten Ergebnis passt.

Bildlich kann man sich das Trainieren wie das Durchprobieren beim Einparken eines Autos in eine Lücke vorstellen: Rechts einschlagen, rückwärts, links einschlagen, Stopp, vorwärts, zu weit, rechts einschlagen, nochmal von vorne, bis es irgendwann passt. Abstand zur Lücke, Lenkeinschlag, Geschwindigkeit usw. wären dann die Gewichtungen. Wie gut man am Ende in der Parklücke steht, würde ein Fehleralgorithmus ermitteln und an den Fahrer senden. Passt das nicht, probiert man nochmal. Weitere Details und weniger hinkende Vergleiche zu Gewichten, bildverarbeitenden Netzen und Verarbeitungsschritten finden Sie in unserem Online-Artikel (Link siehe Kurzinfo).

Kurzinfo

- **Neuronale Netze verstehen**
- **Datenaufbereitung und -aufteilung**
- **Layer für Bilderkennung anwenden**
- **Netz modellieren, trainieren, bewerten und speichern**

Checkliste

Zeitaufwand:
2 Stunden

Kosten:
keine, nur PC erforderlich

Software:
Python und Anaconda Navigator

Mehr zum Thema

- Josef Müller, ESP32CAM liest Wasseruhr, Make 2/21, S. 14
- Daniel Bachfeld, Einstieg in KI, Make 6/18, S. 36
- Josef Müller, KI für den ESP32, Teil 1, Make 6/21, S. 48

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

Vorbereitung

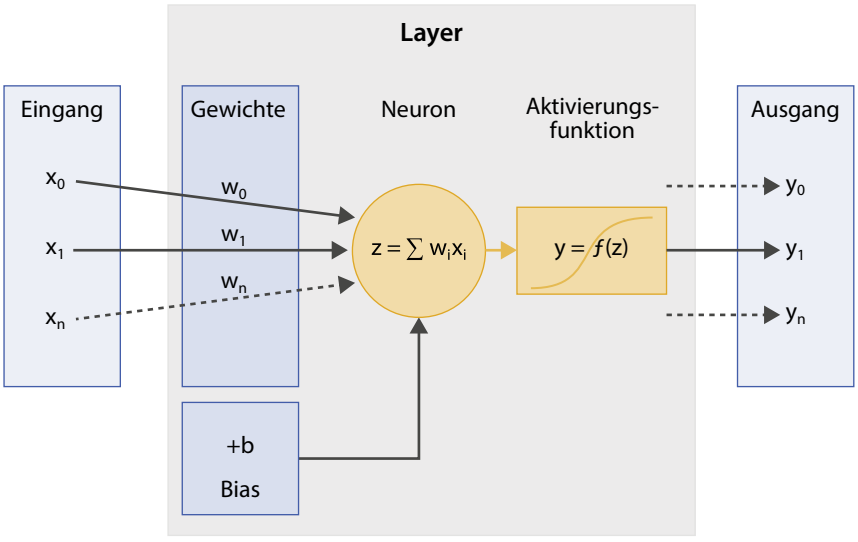
Die Arbeitsschritte bis zu einem fertig trainierten neuronalen Netz für eine Verwendung für den ESP32 haben wir in einzelne Skripte respektive Jupyter-Notebooks unterteilt. Sie finden alle Jupyter-Notebooks für den zweiten Teil auf Github (siehe Link). Die Python-Skripte sind durchnummeriert und bauen aufeinander auf (3). Wie man Jupyter-Notebooks verwendet, haben wir im ersten Teil erklärt.

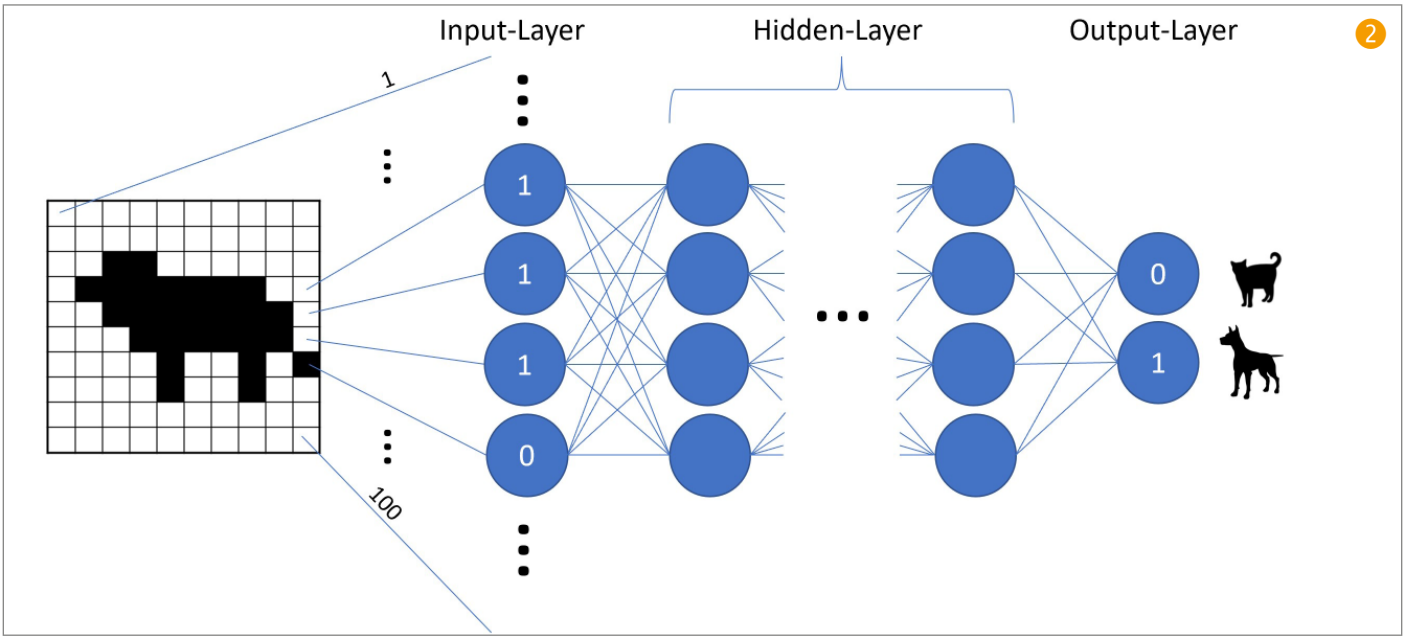
In den dazugehörigen Bildordnern sind bereits zahlreiche Bilder für das Training vorhanden. Eine noch viel größere Anzahl und Variation an Ziffern findet Sie im Trainings-Repository zum Wasserzähler-Projekt (siehe Link).

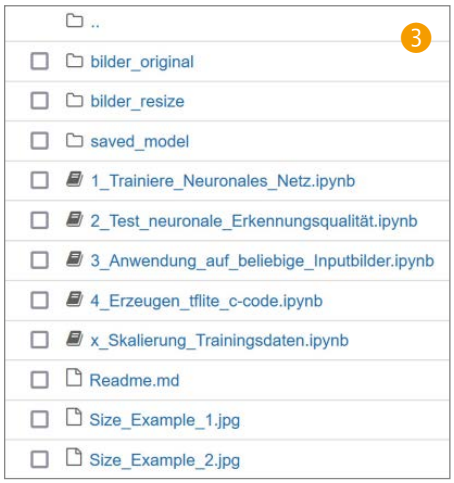
Am Anfang aller Notebooks steht die Variable `ModelNameAndVersion`, in der ein

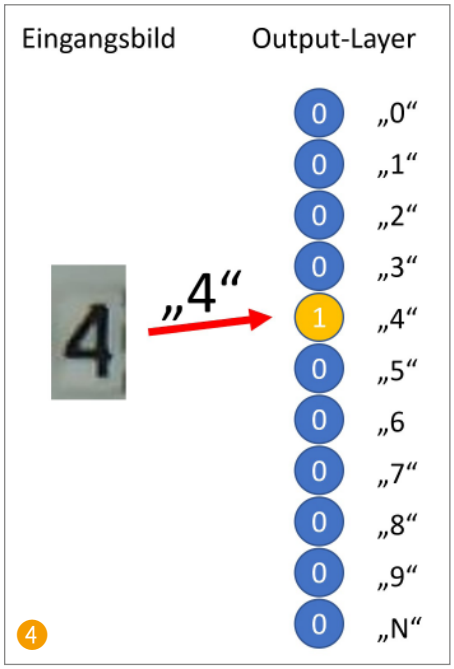
Aufbau eines Neurons (1)

Ein Neuron arbeitet wie ein kleiner Taschenrechner.
Es multipliziert und addiert und gibt sein Ergebnis aus.









eindeutiger Name definiert ist. Unter ihm speichern wir später das neuronale Netz auf dem Dateisystem. Wenn man mit den Notebooks der Reihe nach arbeitet, muss man darauf achten, dass dort immer derselbe Name steht. Wenn man später mit unterschiedlichen Parametern und Optionen experimentieren will, kann man den Namen einfach ändern, ohne das zuvor erstellte und trainierte Netz zu überschreiben.

Netzdefinition

Im ersten Skript wird das neuronale Netz aufgebaut und trainiert. Das Notebook *1_Trainiere_neuronales_Netz.ipnb* folgt folgendem Ablauf:

1. Laden der Bibliotheken und Einstellungen
2. Laden der Bilder
3. Erzeugen der Trainingsdaten & Testdaten
4. Definition und Aufbau des Netzes
5. Training
6. Speichern des neuronalen Netzes

Der erste Schritt lädt alle benötigten Bibliotheken, neben dem Modul *tensorflow* auch *matplotlib* zum Zeichnen von Graphen für die Visualisierung des Trainingfortschritts und der Qualität. Anlehnend an die Skripte für Data Augmentation aus dem letzten Teil werden die Bilddateien geladen. Im Vergleich zum Skript des ersten Artikels muss das hier verwendete um zwei Punkte erweitert werden: Laden und Speichern aller Bilder und Auslesen des Soll-Wertes, also der Ziffer, aus dem Dateinamen.

Wir nehmen an, dass alle Trainingsbilder bereits reskaliert in dem vorhandenen Verzeichnis (`Input_dir = 'bilder_resize'`) vorliegen. Wenn dies nicht der Fall ist, lassen Sie das Notebook *x_Skalierung_Trainingsdaten.ipynb* einfach nochmal laufen, wir haben es in die Skript-Sammlung des zweiten Teils aufgenommen.

Wir nutzen die Python-Systembibliothek *glob*, um die Bilder zu laden. Sie liest alle Dateinamen eines Verzeichnisses und speichert sie in einer Liste, die wir in einer Schleife durch iterieren:

```
files = glob.glob(Input_dir + '/*.
        jpg')
for aktfile in files:
    img = Image.open(aktfile)
    x_data = np.array(img)
    x_data.append(data)
```

Anschließend wird das Bild mittels `Image.open()` geladen, mit der Bibliothek *Numpy* in ein besonders formatiertes Array umgewandelt und in das Datenarray `x_data` geschrieben. respektive angehängt. Das geschieht mit allen Bildern im Verzeichnis, womit die Bilddaten in den Speicher geladen sind. Zusätzlich müssen wir für die Klassifizierung jedem einzelnen Bild noch seine Klasse (Label) zuordnen. Welche Ziffer (Klasse) auf dem jeweiligen Bild zu erkennen ist, ist im ersten Zeichen des Dateinamens angegeben.

Das Zeichen enthält entweder die abgebildete Zahl oder ein N, falls es sich um die Kategorie *Not-A-Number* handelt. Zur Erinnerung: Der Zustand N beschreibt eine uneindeutige Zahl, wenn zum Beispiel das Ziffernrad zwischen zwei Werten steht. Dies wird einem elften Zustand zugeordnet, so dass insgesamt elf Klassen existieren. Das bedeutet im Vorgriff zur Definition des neuronalen Netzes, dass am Ausgangslayer elf Ausgangsneuronen definiert werden, die den einzelnen Klassen entsprechen.

Wenn jetzt also zum Beispiel das Bild eine 4 darstellt, so soll im Idealzustand am Ausgang

nur das Neuron mit der Nummer 4 aktiv sein (=1), alle anderen sollen eine 0 ergeben, inklusive des elften Neurons für Not-A-Number (NaN).

Da dies eine typische Aufgabe für Klassifizierungsnetzwerke ist, gibt es in Tensorflow eine Funktion, die aus einer Zahl einen Vektor erzeugt, der genau diesem Schema entspricht:

```
category_vektor = tf.keras.utils.to_
categorical(category, 11)
```

Dieser Zielvektor (4) wird in einem weiteren Array `y_data` beigefügt, wobei die Indizes der beiden Arrays dann jeweils auf die zusammengehörenden Daten zeigen. Im Ergebnis hat man nun zwei Arrays, deren erster Index jeweils über die Trainingsbilder iteriert und die entweder die Bilddaten als Array ($20 \times 32 \times 3$) oder den Zielzustand (11) enthalten. Das Skalar 3 in $20 \times 32 \times 3$ bezieht sich auf den Farbraum der Bilder: je 1 Byte für R, G und B (3 Bytes, zusammen 24 Bit). Bei der Zuordnung der Zielzustände ist wie bei allen Arrays zu beachten, dass das 11. Element für N der Klasse 10 zugeordnet ist – wir fangen ja bei 0 an zu zählen.

Trainings- und Testdaten

Um den Trainingserfolg unseres Notebooks zu messen, ist es notwendig, dass man mindestens zwei Datensätze hat: einen Trainingsdatensatz, mit dem das Netz trainiert wird, und einen zweiten Satz an Evaluierungsdaten, an dem man die Erkennungsleistung misst.

Ansonsten besteht die Gefahr, dass das neuronale Netz zwar auf den ersten Blick immer besser wird und seine Fehler stetig abnehmen. In Wahrheit lernt es aber alle Bilder eines Trainingssatzes einfach auswendig. Sobald man dann untrainiertes Bildmaterial verwendet, sinkt die Erkennungsleistung erheblich.

Da wir nur einen Datensatz haben, teilen wir ihn einfach auf. Praktischerweise hat Tensorflow dafür eine eingebaute Funktion. Da die Bilder aufgrund des Dateinamens in fester Reihenfolge geladen wurden, müssen die Bilder aber zunächst einmal zufälliger angeordnet werden. Dazu verwenden wir die Funktion `shuffle()` des am Anfang des Notebooks eingebundenen Moduls *Sklearn*. Zwar ändert sich damit die Reihenfolge, die Zuordnung von Bild und Ergebnis (Klasse) in beiden Arrays bleibt aber erhalten. Anschließend kann man über die Funktion `train_test_split()` das Array in einem vorgegebenen Verhältnis aufteilen und hat somit die beiden Datensätze. Wir teilen den Datensatz zu 80 Prozent Training und 20 Prozent Evaluierung auf:

```
x_data, y_data = shuffle(x_data, y_
data)
Training_Percentage = 0.2
x_train, x_test, y_train, y_test =
train_test_split(x_data, y_data,
test_size=Training_Percentage)
```

```
datagen = ImageDataGenerator(width_shift_range  = [-Shift_Range, Shift_Range],
                             height_shift_range = [-Shift_Range, Shift_Range],
                             brightness_range   = [1-Brightness_Range, 1+Brightness_Range],
                             zoom_range         = [1-ZoomRange, 1+ZoomRange],
                             rotation_range     = Rotation_Angle)
Batch_Size = 4
train_iterator      = datagen.flow(x_train, y_train, batch_size=Batch_Size)
validation_iterator = datagen.flow(x_test,  y_test,  batch_size=Batch_Size)
```

(5)

Im Training wird wieder Data Augmentation mit der Funktion `datagen = ImageDataGenerator()` eingesetzt, um die Bildersammlung künstlich zu erweitern. Bei den möglichen Variationen wurden im Notebook gut funktionierende Standardwerte verwendet. Sie können hier gerne mit Variationen spielen, um zu sehen, was am besten für Sie funktioniert.

Um später das neuronale Netz mit den Bildern zu füttern, benutzt man sogenannte Iteratoren. Ein Iterator funktioniert unter Python im Prinzip wie ein Nummern-Zieh-Automat in Behörden. Wenn man den Knopf drückt, spuckt der Automat einen Zettel mit der nächsten Wartenummer aus.

Das funktioniert unter Python mit beliebigen Objekten: Immer wenn man den Iterator abfragt, gibt er das nächste Objekt aus seiner Liste zurück, hier Bilder (5). Genau genommen ist es hier immer eine Sammlung von Bildern, nämlich ein Batch. Der Parameter `Batch_Size` legt im Notebook fest, wieviele Bilder geliefert und in einem Lernzyklus des neuronalen Netzes gleichzeitig bewertet werden sollen, bevor die Gewichte angepasst werden sollen.

Da man zwei Datensätze hat, werden zwei Iteratoren im Notebook definiert – einen für die Trainings- und einen für die Testdaten.

Datenreduktion

Für Bilderkennungsaufgaben besteht ein neuronales Netz meist aus einer Abfolge von *Convolutional*- und *Pooling*-Layern (siehe Einführungs-Artikel unter dem Link) und daran haben wir uns auch hier orientiert. Die Anzahl und Größe der Layer hängen wiederum von den Eingangsdaten ab, hier der Größe der Bilder und den zugeordneten Klassen. Im Vorfeld haben wir einige verschiedene Optionen getestet. Im Folgenden zeigen wir einen Netzaufbau, der sich für diese Aufgabe bewährt hat.

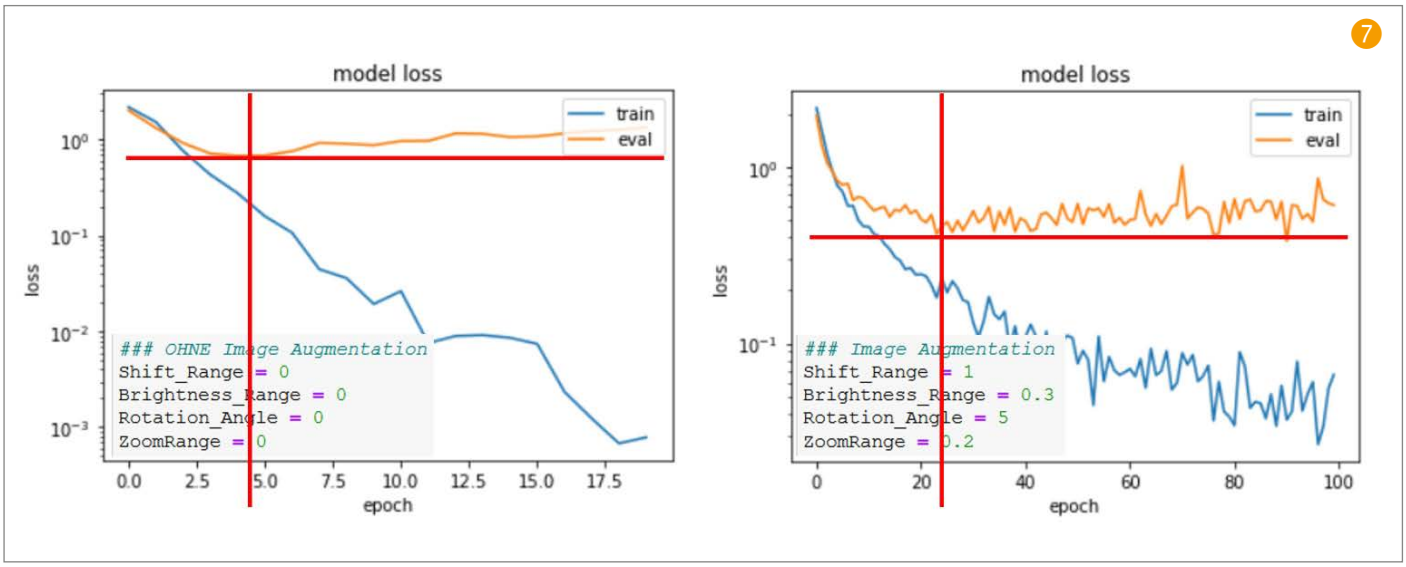
Im Eingangslayer findet zunächst eine Normalisierung der Bilddaten statt (6). Dies wird immer dann angewandt, wenn der Wertebereich nicht gut zum optimalen Wertebereich der Neuronen passt. Die Farbinformationen in Bilddaten geht typischerweise pro Farbkanal von 0 bis 255, Neuronen rechnen vereinfacht gesagt aber lieber mit kleineren Werten zwischen 0 und 1.

Dann wird dreimal hintereinander die Kombination aus *Faltungs-* und *Pooling-Layer* angewendet (Details zu diesen Layern finden Sie unter dem Link). Die Funktion `Conv2D` definiert die Faltung, wobei der erste Parameter die Anzahl der *Featuremaps* angibt und der anschließende Vektor die Größe des Operatorkerns. Als letzter interner Layer wird ein einzelner flacher vollverbundener Layer eingefügt (*Flatten*). Er macht quasi aus der Matrix des letzten Pooling-Layers eine lange Kette von Neuronen.

Der Ausgang besteht aus den oben beschriebenen elf Zuständen. Die Aktivierungsfunktion `softmax` sorgt dafür, dass die Ausgangswerte aller Neuronen zwischen 0 und 1 liegen und in der Summe genau 1 ergeben. Man kann also den Wert jedes einzelnen Ausgangs-Neurons als Wahrscheinlichkeit interpretieren, dass das Bild dessen zugeordnetem

```
model = tf.keras.Sequential()

model.add(BatchNormalization(input_shape=(32,20,3)))

model.add(Conv2D(16, (3, 3), padding='same', activation="relu"))
model.add(MaxPool2D(pool_size=(2,2)))
model.add(Conv2D(32, (3, 3), padding='same', activation="relu"))
model.add(MaxPool2D(pool_size=(2,2)))
model.add(Conv2D(32, (3, 3), padding='same', activation="relu"))
model.add(MaxPool2D(pool_size=(2,2)))
model.add(Flatten())
model.add(Dense(256,activation="relu"))

model.add(Dense(11, activation = "softmax"))

model.summary()

model.compile(loss= tf.keras.losses.categorical_crossentropy,
              optimizer= tf.keras.optimizers.Adadelta(learning_rate=0.6, rho=0.95),
              metrics = ["accuracy"])
```

(6)



Wert entspricht. Bei einer 4 liefert beispielsweise das dazugehörige Neuron einen Wert von 0,99 während alle anderen Neuronen 0,01... liefern.

Die Vorhersage des neuronalen Netzes erhält man, indem man einfach nach dem Neuron mit dem höchsten Wert sucht. Ist es das erste Neuron, so entspricht es der 0, beim zweiten der 1, usw. bis zum zehnten Neuron für die 9 und dem elften Neuron für den Zustand *Not-A-Number* und die Klasse 10.

Über die Funktion `model.summary()` bekommt man einen Überblick über das neuronale Netz und auch die Anzahl der zu trainierenden Parameter. Als letzten Schritt muss die Struktur noch kompiliert werden, dies ist gleichbedeutend mit dem Aufbau des Netzes im System, so dass es anschließend trainiert und verwendet werden kann. Die in der `compile()`-Funktion übergebenen Parameter beziehen sich auf bestimmte Verfahren zu Minimierung des Fehler respektive zur Optimierung während der Trainingsphase.

Training

Das eigentliche Training ist nach all diesen Vorbereitungen sehr einfach. Es besteht eigentlich nur aus einem einzigen Befehl mit der Funktion `model.fit()`. Ihr übergibt das Skript die Trainings- und Evaluierungsdaten sowie die Anzahl der *Epochen*, die man trainieren möchte. Eine Epoche entspricht genau einem Umlauf über alle Trainingsdaten. Dies ist der rechenaufwändigste Schritt und er kann je nach Rechenpower und Anzahl der Bilder viel Zeit in Anspruch nehmen. Die Anzahl der Testbilder und die Größe des hier gewählten Netzes sollte jedoch ein Training auf nahezu allen üblichen Rechnern ermöglichen.

Der Rückgabewert der Trainingsfunktion liefert ein Array über den Trainingsverlauf. Darin wird der Fehler (die Loss-Funktion) sowohl der Trainings- wie auch der Testdaten für jede Trainingsepoche gespeichert.

Eine grafische Darstellung über `plt`-Funktion ermöglich einen guten ersten Eindruck 7. Die Fehler sollten zunächst mit jeder Trainingsepoche abnehmen. Um zu erkennen, wie viele Epochen sinnvoll sind, kommt jetzt der Unterschied zwischen Trainings- und Evaluierungsdaten zum Tragen. Meistens kann man beobachten, dass die Fehler bei den Trainingsdaten noch sehr lange abnehmen.

Da die Evaluierungsdaten nicht zum Anpassen der Gewichtungen verwendet werden, können diese auch nicht gelernt werden. Daher ist deren Fehlerquote ein gutes Maß für das Lernvermögen. Zu Beginn nimmt der Fehler auch hier noch mit fortschreitender Epochenzahl ab. Ab einem bestimmten Punkt stagniert dies jedoch. Dies ist ein klares Zeichen dafür, dass jetzt keine weiteren allgemeinen Eigenschaftes der Trainingsbilder gelernt werden.

Erkennungsqualität

Wenn die Variationen in den Bildern sehr gering sind, so kommt es auch vor, dass die Fehler für die Testdaten sogar wieder zunehmen, obwohl die Fehler für die Trainingsdaten noch weiter zunehmen. Ab diesem Punkt wird die Gesamtperformance des Netzwerkes in der Regel wieder schlechter, denn die Gewichte beginnen, sich sehr spezifisch an die individuellen Trainingsdaten anzupassen, sie quasi auswendig zu lernen. Ein längeres Training ist nicht mehr sinnvoll.

Dank Data Augmentation und den zusätzlich variierenden Eingangsbildern tritt dies erst sehr spät auf. Die Kurven in 7 zeigt ein solches Verhalten im Vergleich. Um das Verhalten ohne Data Augmentation zu beobachten, ist es am einfachsten, die Parameter der Bildvariation auf 0 zu setzen. Man sieht, dass nach rund vier Epochen der Fehler in den Testdaten wieder zunimmt. Aber auch mit Augmentation erkennt man, dass bei diesem Datenset nach ca. 20 Trainingsepochen kein signifikanter Fortschritt in den Evaluierungsdaten mehr zu erkennen ist.

Um die trainierten Eigenschaften des neuronalen Netzes dauerhaft zur Verfügung zu haben, wird im letzten Schritt das neuronale Netz gespeichert:

```
model.save('saved_model/' +
ModelNameAndVersion)
```

Das Model wird vollständig (Aufbau, Trainingsergebnis) im H5-Format (*Hierarchical Data Format 5*) gespeichert. Da es aus mehreren Dateien besteht, wird es unter dem am Anfang definierten Namen in dem Unterverzeichnis *saved_model* gesichert. Von dort kann es dann in den späteren Schritten wieder geladen werden, ohne dass das Training erneut durchgeführt werden muss.

Die Fehlerfunktion liefert zwar ein gutes Indiz, ob das neuronale Netz Epoche für Epoche noch lernt, aber daraus kann man nur schlecht direkt auf die Qualität der Erkennungsrate schließen. Einen besseren Eindruck von der Qualität kann man bekommen, wenn man zunächst erst einmal das trainierte Netz auf die ursprünglichen Bilder anwendet. Im zweiten Skript *2_Test_neuronale_Erkennungsqualität.ipynb* wird gezeigt, wie man ein trainiertes gespeichertes Netz wieder zur Verfügung stellt und anschließend dieses Netz auf die ursprünglichen Trainingsdaten anwendet.

Anwendung

Das gespeicherte Netz kann genauso einfach geladen werden, wie es auch gespeichert wurde:

```
model = tf.keras.models.load_
model('saved_model/' +
```

```
ModelNameAndVersion)
model.summary()
```

Die zweite Zeile zeigt einen Überblick und man erkennt an der Ausgabe die vorab definierte Netzstruktur. Somit ist man jetzt in demselben Status wie nach dem direkten Training des Netzes. Man könnte sogar das Training von hier aus weiterführen.

Im zweiten Teil dieses Skriptes werden die Trainingsbilder durch das neuronale Netz geschickt. Ähnlich zum Laden der Trainingsbilder wird auch hier über alle Dateien in dem Ordner mit den normierten Bildern ($20 \times 32 \times 3$) iteriert. Auch wird zunächst aus dem ersten Zeichen des Dateinamens der erwartete Zahlenwert bestimmt und in `Classification_SOLL` hinterlegt. Als nächstes wird das Bild geladen. Die Erkennung des Bildes über das neuronale Netz erledigt eine einzelne Zeile:

```
result = model.predict(img)
```

Die Funktion `model.predict()` liefert als Ergebnis den Output-Layer zurück, also ein Array (Vektor) mit elf Wahrscheinlichkeiten. Glücklicherweise gibt es eine Funktion in der Bibliothek *Numpy*, die die Klasse mit dem höchsten Wert zurückgibt:

```
classes = np.argmax(result, axis=-1)
Classification_IST = classes[0]
```

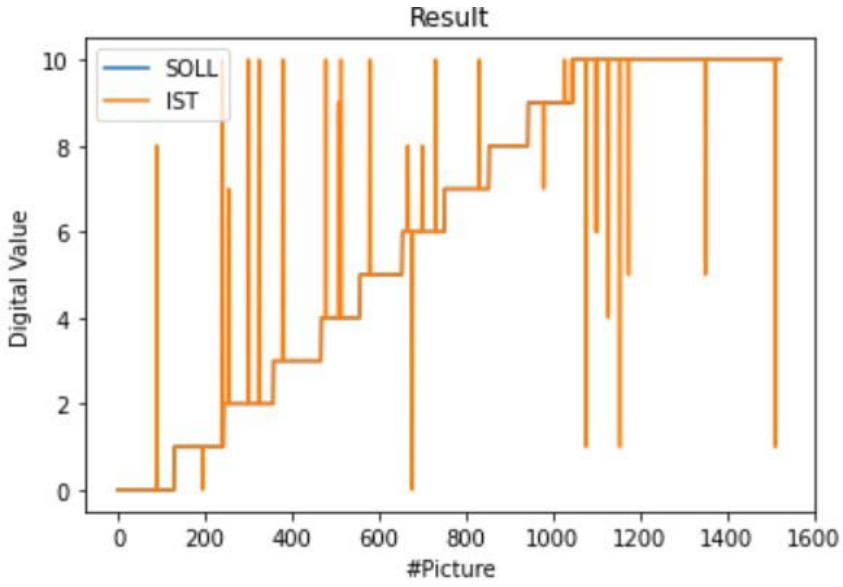
Die erste Funktion findet das Maximum, liefert dieses aber in einem 1-dimensionalen Array, welches dann in der zweiten Zeile der Ergebniszahl `Classification_IST` zugewiesen wird.

Im Rest des Notebooks wird zum einen der Soll- und der Ist-Zustand in einem Array abgelegt. Zum anderen wird bei einer Abweichung zwischen Soll und Ist eine Ausgabe mit den Ergebnissen und dem zugehörigen Bild erzeugt. Dies ist sehr nützlich, um einen Eindruck der nicht erkannten Bilder zu bekommen. Oft erkennt man schon an den falsch erkannten Bildern eine Systematik, wie zum Beispiel sehr geringer Kontrast oder ähnliches.

Die abschließende Visualisierung 8 zeigt die Abweichung zwischen Soll- und Ist-Werten und gibt einen guten Überblick, welche Anzahl an Bildern falsch klassifiziert wurden. Die 0 wurde beispielsweise einmal fälschlicherweise als 8 erkannt. Die 1 einmal als 0. Die 2 wurde einmal als 7 erkannt, zweimal wurde sie gar nicht erkannt, also jeweils 10 für NaN.

Man darf nicht die Erwartung haben, dass Bilder immer fehlerfrei klassifiziert werden. Selbst große und aufwendig trainierte neuronale Netze haben eine Erkennungsrate deutlich kleiner als 100 Prozent. Daher sollte die Verwendung von Ergebnissen aus neuronalen Netzen auch immer weiteren Plausibilitätskontrollen oder Fehlerkorrekturmechanismen unterliegen. Wird etwa bei der Wasseruhr eine Ziffer nicht erkannt, so wird zunächst versucht, diese aus dem vorherigen Wert abzuleiten.

8



Wenn dies auch nicht sinnvoll gelingt, wird der gesamte Wert verworfen. Dies hat sich als sehr effektiv und nützlich herausgestellt. Fehlertolerante Konzepte sind ein wesentlicher Teil einer Anwendung von neuronalen Netzen.

Das Skript *3_Anwendung_auf_beliebige_Inputbilder.ipynb* ist eine geringfügige Modifikation des vorherigen. Der Unterschied besteht darin, dass nun ein einzelnes beliebiges Bild geladen, skaliert und erkannt wird. Sie müssen es dazu im Ordner der Jupyter-Notebooks ablegen und gegebenenfalls den Dateinamen im Skript anpassen.

ESP32-Verwendung

Wir haben nun ein Modell, das auf dem PC in Python gut funktioniert. Um es auch auf dem ESP32 zu verwenden, muss es quasi übersetzt werden. Die Tensorflow-Bibliotheken, die später auf dem ESP32 verwendet werden, sind speziell angepasste C-Implementierungen. Sie sind besonders kompakt und auf die Ausführung auf kleineren CPUs mit geringem Speicher ausgelegt.

Dafür ist auch ein spezielles Speicherformat vorgesehen, welches sich *tflite* nennt. Darin ist die gesamte Information zu Netzaufbau und den trainierten Gewichten in einer einzigen durchgängigen Struktur in einem binären Datenformat abgelegt. Diese Binärdaten können nachher einfach mit einer *tflite*-Funktion geladen werden und das neuronale Netz ist quasi einsatzbereit.

Hier sollen zwei Möglichkeiten gezeigt werden, mit denen man das Netz in dem kompakten Format speichert und später im Quellcode verwenden kann:

1. Die Speicherung als C-Datenobjekt zur Einbindung im Quellcode oder
2. Die Speicherung als Datei, um es dynamisch von einer Datenquelle zu laden.

Wie das funktioniert, ist im Notebook *4_Erzeugen_tflite_c-code.ipynb* gezeigt. Die Umwandlung in das tflite-Format wird mittels eines Konverters durchgeführt:

```
converter    =
tf.lite.TFLiteConverter.from_keras_
model(model)

tflite_model = converter.convert()
```

Im Datenobjekt `tflite_model` steht nun die binäre Codierung unseres Netzes. Diese kann einfach in eine Datei geschrieben werden:

```
open(ModelNameAndVersion + ".tfl",
"wb").write(tflite_model)
```

Ich verwende hier typischerweise die Endung *.tfl* oder *.tflite*.

C-Datenobjekt

Wenn man im späteren Programm keine Dateien dynamisch verwenden kann oder möchte, so besteht auch die Möglichkeit, das tflite-Model in einer C-Headerdatei direkt als `unsigned char`-Array zu definieren und in der Firmware direkt mit zu kompilieren. Dazu wird das Binärobjekt mittels einer Hilfsfunktion Byte-weise in Hexcode umgewandelt und im C-Headerformat codiert. Am Ende wird noch die Größe des Arrays in einer gesonderten Variablen gespeichert und das ganze dann als h-File, also im Klartext und nicht als Binärtyp-Datei, gesichert.

Im kommenden dritten Teil unserer Serie zeigen wir dann, wie man das Netz in den C-Code für ESP32CAM einbindet und direkt auf dem ESP32 und Bilder von Ziffern klassifiziert. *—dab*