c't Heft 12/2021 S. 130-135 / Wissen - TensorFlow auf 4 Raspis

Lerngruppe Montagsmaler

Verteiltes maschinelles Lernen mit TensorFlow auf einem Raspi-Clusterchen

Die Rechenlast fürs Training neuronaler Netze kann TensorFlow ab Version 2.3 auf mehrere Rechner verteilen. Dafür muss man nicht gleich einen Cluster in der Cloud mieten. Ein paar Raspis genügen.

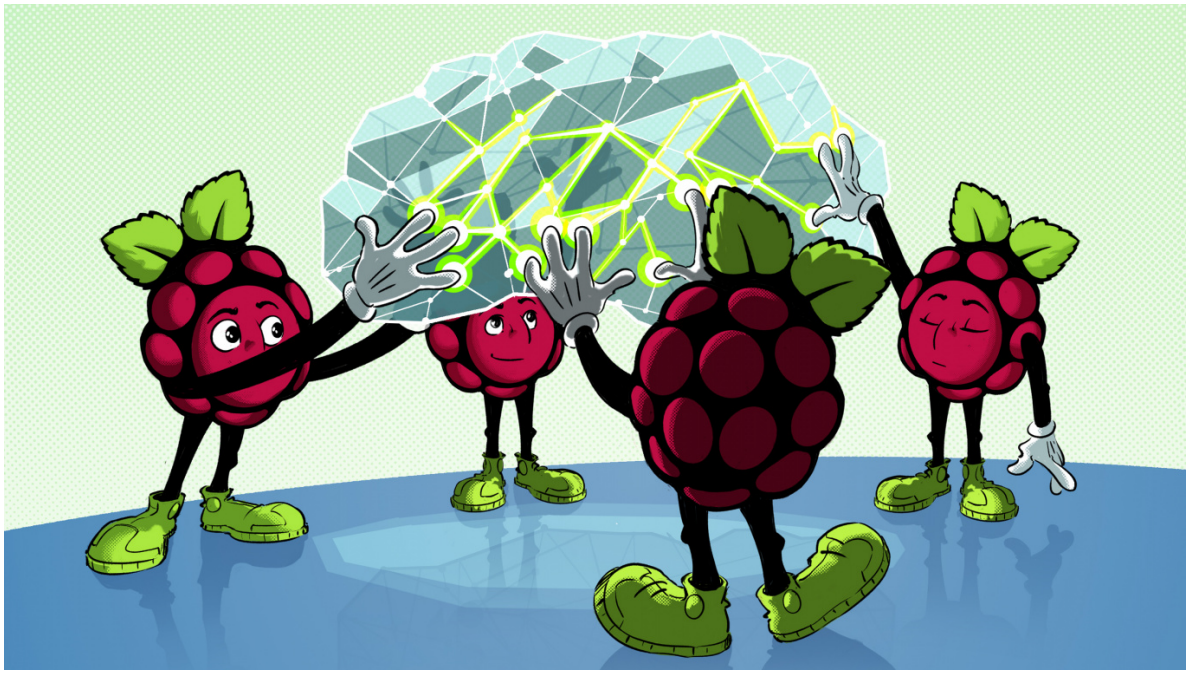


Bild: Albert Hulm

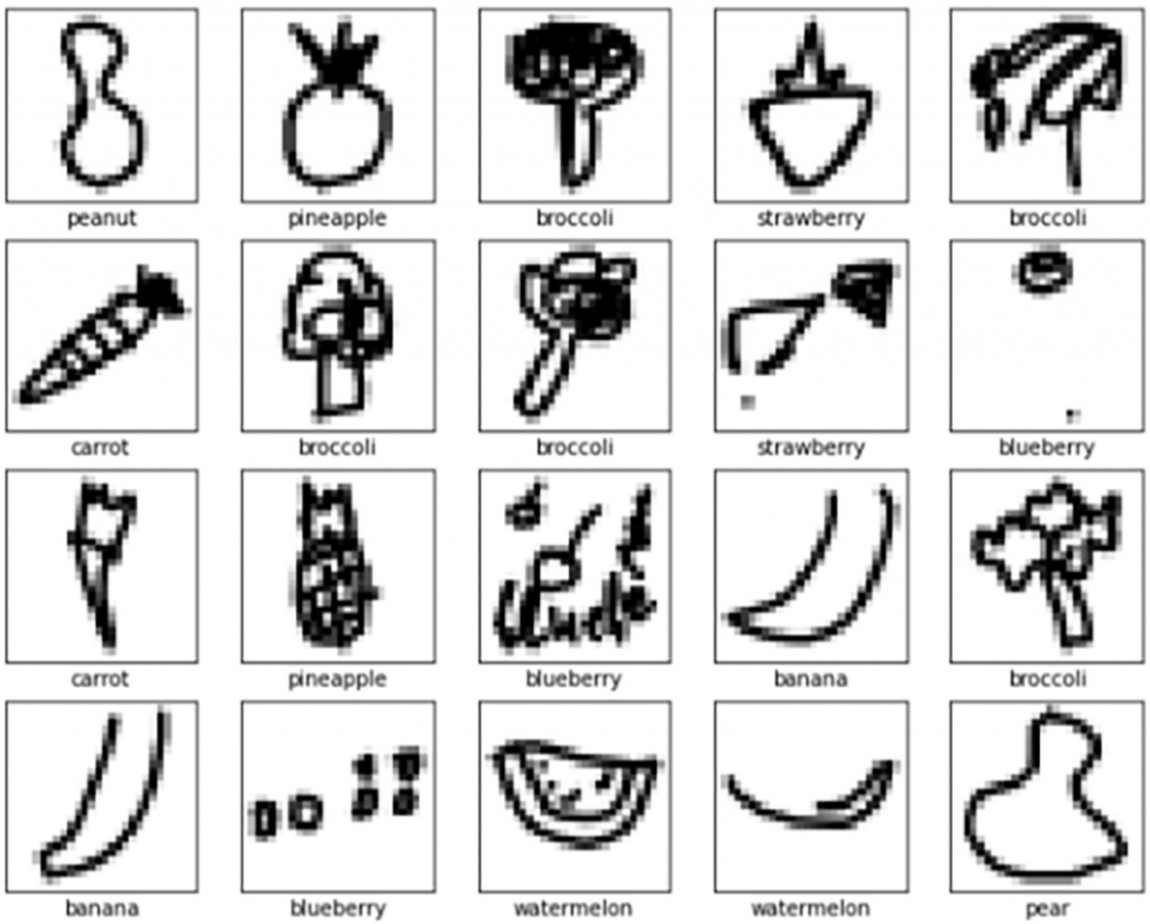
Das Training neuronaler Netze verschlingt so viel Rechenzeit, dass mit einer einzelnen CPU auf der Entwicklermaschine nur kleine KI-Experimente gelingen. Etwas größer darf das Netz werden, wenn man eine Grafikkarte verwendet, die vom KI-Framework TensorFlow unterstützt wird. Wer aber richtig große Netze lernen lassen will, muss Cluster mit vielen Rechnern für die Aufgabe zusammenschalten. Angemietete Cloudrechner, im Idealfall mit Spezialhardware wie Googles Tensor Processing Units (TPU), sind optimal. Die kosten aber pro Stunde. Die Abrechnung nach Zeit schreckt gerade in der Anfangsphase eines KI-Projekts ab, wenn Debugging und Konfiguration noch nicht abgeschlossen sind. Eine günstigere Alternative, um das Lernen im Cluster auszuprobieren, findet sich in vielen Schubladen: Die KI-Algorithmen von TensorFlow laufen auch auf mehreren Raspberry Pis. Mit denen gelingt der Einstieg in die komplexe Welt hochparallelen Rechnens ohne viel Verdrahtung; als Vorwissen reichen Grundkenntnisse mit Keras [1]. Zwei oder mehr übers WLAN vernetzte Minirechner genügen zum Üben (wir haben mit vier Raspis getestet) und verdeutlichen das Prinzip, ersetzen aber kein Rechenzentrum.

c't Hardcore

Was die vier Raspberry Pis in diesem Artikel lernen sollen, ist das Erkennen von mit Hand gezeichneten Bildern. Die Aufgabe ähnelt dem Spiel Montagsmaler: Ein Spieler bekommt einen beliebigen Begriff und zeichnet diesen mit möglichst wenigen Strichen. Der andere, in diesem Fall die Raspberry Pis, müssen erkennen, um welchen Begriff es sich handelt.

Die für das Lernen notwendigen Trainingsdaten stammen aus der Anwendung Quick

Draw von Google. Unter der URL quickdraw.withgoogle.com kann jeder einmal selbst Begriffe zeichnen und sehen, wie gut die Montagsmaler-KI von Google die Bilder erkennt.



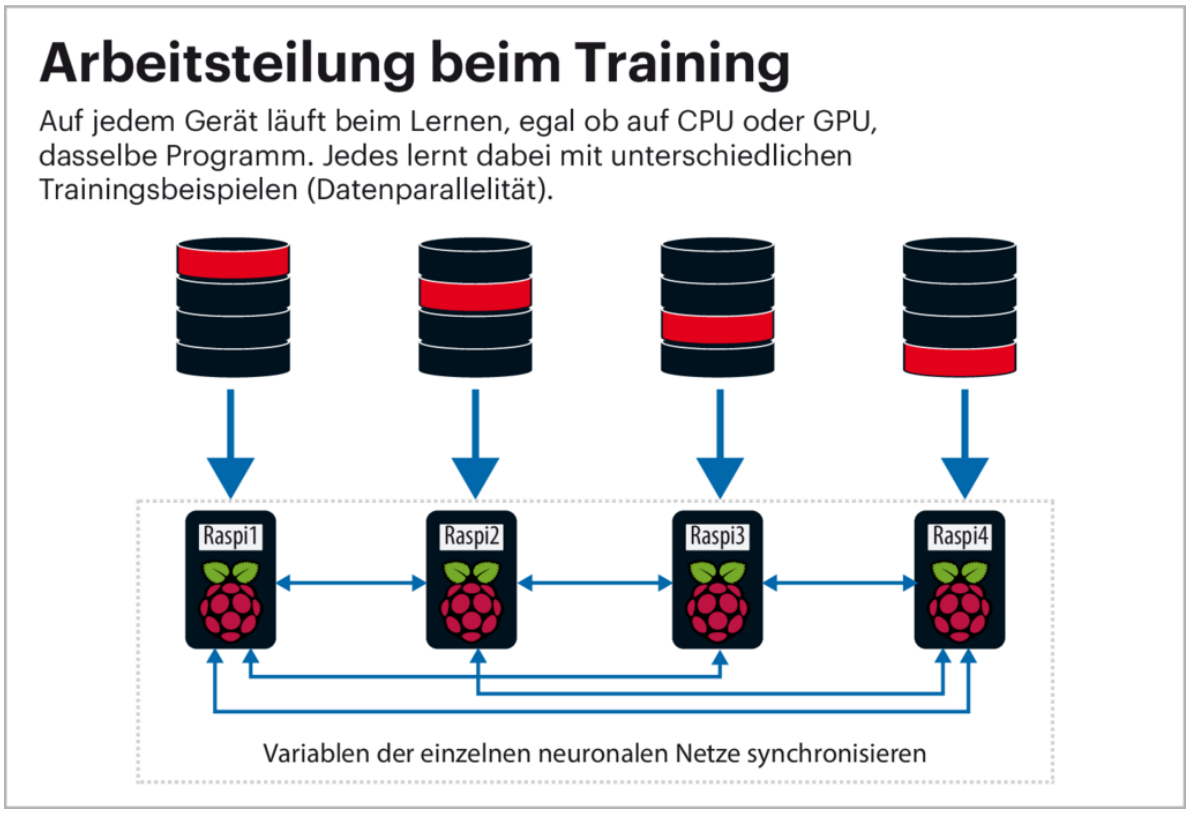
Anhand einfacher Strichzeichnungen sollen die Raspberry Pis die dargestellten Begriffe erkennen.

Der gesamte Beispieldatensatz von Google ist über 50 Gigabyte groß - zu viel für kleine Raspis. Das Beispielprogramm lernt daher nur Daten aus dem Bereich Obst und Gemüse, 13 Begriffe.

An solchen Einschränkungen merken Sie: Der Raspberry Pi ist keine optimierte KI-Hardware.

Datenparalleles Lernen

Fürs verteilte Lernen gibt es zwei mögliche Strategien: Entweder verteilt das Framework die Daten an mehrere Rechner, die jeweils ihren Anteil durchrechnen und die Lernsignale danach zusammenwerfen, damit jeder Rechner von den Erkenntnissen der anderen Rechner profitiert (Datenparallelität). Alternativ zerschneidet das Framework das Netzwerk in Teile, und jeder Rechner berechnet seinen Teil mit allen Daten (Modellparallelität). Bei letzterer Methode müssen sich die Rechner gegenseitig viele Zwischenergebnisse zusenden, was viel schnelle Netzwerkkommunikation nötig macht. Da die Raspis der Einfachheit halber über WLAN kommunizieren, ist das Netzwerk ein Flaschenhals und datenparalleles Rechnen bietet sich an. TensorFlow bringt diese Strategie im Modul *tensorflow.distribute.Strategy* mit.



Auf jedem der Rechner läuft der Lernvorgang folgendermaßen ab: Das vollständig im Speicher liegende neuronale Netz bekommt als Eingabe einen Anteil der Trainingsbeispiele (in diesem Fall einige Zeichnungen) und wirft als Ergebnis den Begriff aus, um den es sich handeln könnte. Ist das Ergebnis nicht ganz richtig, geht der Optimierungsalgorithmus rückwärts durch das Netzwerk und registriert, welche Gewichte welcher Neuronen wie stark zum Fehler beigetragen haben (Backpropagation of Errors, Erklärung siehe [2]). Mathematisch sind das die Gradienten der Loss-Funktion. Mit diesen Gradienten justiert der Lernvorgang von TensorFlow die Variablen des neuronalen Netzes nach, damit beim nächsten Mal etwas Besseres herauskommt.

Sind am datenparallelen Rechnen mehrere Geräte beteiligt, ändert sich nur der letzte Schritt: Da jeder Rechner mit anderen Beispielen trainiert hat, haben alle etwas andere Gradienten berechnet. Man könnte auch sagen: Jeder Rechner hat eine etwas andere Idee davon, mit welchen Änderungen sich das Netzwerk verbessern ließe. Die Wahrheit liegt wie üblich dazwischen: Die Rechner tauschen ihre Verbesserungsvorschläge (Gradienten) aus und berechnen den Durchschnitt. So profitiert jeder von den Erfahrungen der anderen und bewertet Ausreißer im eigenen begrenzten Datensatz nicht über.

Etwas technischer ausgedrückt kombiniert der Optimierungsalgorithmus (eine Form des Gradientenabstiegs) die Gradienten der Neuronengewichte und verteilt diese an alle beteiligten Rechner. Sobald diese mit diesen Durchschnittsgradienten die Gewichte des Netzwerks aktualisiert haben, können sie mit dem nächsten Happen an Trainingsbeispielen (Batch) den nächsten Trainingsdurchlauf starten. Das geht so weiter, bis der gesamte Lernvorgang nach $32 \cdot 256 = 8192$ Batches beendet ist.

Diese Strategie nennt TensorFlow *MirroredStrategy* . Sie ist für Grafikkarten mit mehreren Rechenkernen gedacht. Jeder dieser Kerne hat eine Kopie (mirror) der Variablen des neuronalen Netzes. TensorFlow wartet nach jedem Lerndurchgang, bis alle GPU-Programme sich melden, dann kombiniert es die Werte der einzelnen Meldungen und verteilt diese Werte wieder an alle. Das findet aber auf einem Rechner mit vielen Kernen statt.

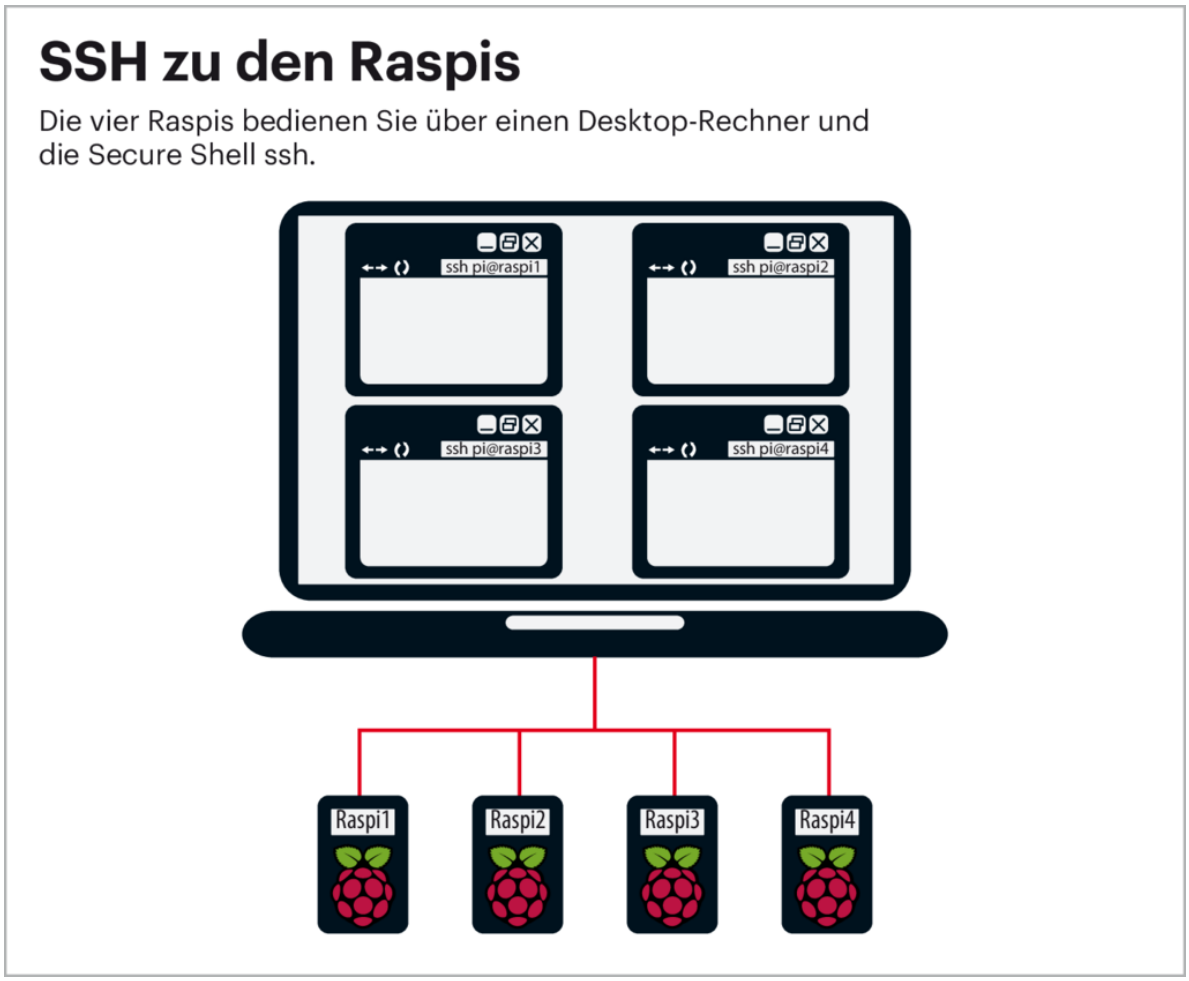
Eine davon abgeleitete Strategie ist die *MultiWorkerMirroredStrategy* . Sie läuft sehr ähnlich ab wie die *MirroredStrategy* , allerdings können unterschiedliche Rechner daran teilnehmen. Die Kommunikation zwischen den Geräten läuft dann übers Netzwerk. Daher ist die *MultiWorkerMirroredStrategy* für das Lernen mit zwei oder mehr Raspis (oder Cloud-Servern) geeignet.

Raspberry-Pi-Installation

Der Datensatz und das neuronale Netz belegen viel Arbeitsspeicher. Um mit dem begrenzten Platz auf dem Raspberry Pi möglichst sparsam umzugehen, sollten Sie

daher nur die nötigste Software installieren. Raspberry Pi OS Lite (32 Bit) ohne Bedienoberfläche bringt ungefähr 0,4 GByte auf die Waage, mit voller Ausstattung sind es dagegen 2,5 GByte.

Außerdem ist es wenig sinnvoll, bei vier Raspis mit vier Bildschirmen, vier Tastaturen und vier Mäusen arbeiten zu wollen. Daher bieten sich sogenannte "Headless"-Installationen an, bei der man von einem Desktop-Rechner aus die Raspis installiert und steuert.



Am schnellsten geht die Installation mit dem "Imager" aus dem Download-Bereich von raspberrypi.org. Dieses Programm schreibt das gewählte Raspberry OS auf eine SD-Karte. Da Sie keine Oberfläche benötigen, wählen Sie "Raspberry Pi OS (other)/Raspberry Pi OS Lite (32 Bit)". Mit der Tastenkombination Strg+Umschalt+X öffnet sich ein Menü, in dem Sie das Passwort, die WLAN-Daten und den Hostnamen des neu installierten Systems einstellen und die Secure Shell (*ssh*) mit einem Haken aktivieren können [3].

Paramiko

Um alle Raspis einheitlich zu konfigurieren, hilft die Python-Bibliothek Paramiko. Dieser Python-Wrapper um *ssh* führt programmierbare Befehle auf allen Raspis aus. Installieren Sie dafür zunächst auf dem Desktop-PC die Bibliothek:

```
pip install paramiko
```

Im Repository, das Sie über ct.de/ykkz finden, liegt das Skript *install_pi.py* , das das System auf dem Raspi aktualisiert. Es funktioniert folgendermaßen: Zunächst importiert es den *SSHClient* aus der Bibliothek, erzeugt danach ein Objekt dieser Klasse und lädt die im System hinterlegten Schlüssel:

```
from paramiko.client import SSHClient
client = SSHClient()
```

```
client.load_system_host_keys()
```

Danach folgt die Liste der Hostnamen, mit denen sich Paramiko nacheinander verbinden soll:

```
server_list = ['raspi1', 'raspi2',
               'raspi3', 'raspi4']
```

Die Funktion *for_all()* führt mit *client.exec_command()* Befehle auf jedem Rechner aus *server_list* aus und schreibt die Ausgaben der Befehle jeweils auf die Konsole vom Desktop:

```
def for_all(command):
  for i in server_list:
    print(i)
    client.connect(i, username='pi')
    stdin, stdout, stderr = \
          client.exec_command(command)
    for line in stdout:
      print('... ' + line.strip('\n'))
    for line in stderr:
      print('... ' + line.strip('\n'))
    client.close()
```

Danach reichen für ein Update des Systems auf allen Raspis die folgenden zwei Funktionsaufrufe:

```
for_all('sudo apt-get update')
for_all('sudo apt-get -y upgrade')
```

TensorFlow installieren

Für den ARM-Prozessor im Raspberry Pi gibt es leider kein offizielles Binärpaket von TensorFlow. Statt wie in der TensorFlow-Doku beschrieben das Framework selbst zu kompilieren, können Sie die inoffiziellen Binärpakete von GitHub-User PINTO0309 nutzen. Der beschreibt unter github.com/PINTO0309/Tensorflow-bin/, wie Sie das Paket installieren. Halten Sie sich an die Anleitung für das aktuelle TensorFlow 2.x in Kombination mit Python 3. Wir haben mit den folgenden Befehlen TensorFlow 2.4.0 mit sämtlichen Abhängigkeiten auf einem Raspi 3 und einem Raspi 4 installiert (*install_pi.py* im Git-Repository):

```
sudo apt-get install -y libhdf5-dev \
    libc-ares-dev libeigen3-dev gcc \
    gfortran python-dev libgfortran5 \
    libatlas3-base libatlas-base-dev \
    libopenblas-dev libopenblas-base \
    libblas-dev liblapack-dev cython \
    libatlas-base-dev openmpi-bin \
    libopenmpi-dev python3-dev \
    python3-pip
sudo pip3 install \
   keras_applications==1.0.8 --no-deps
sudo pip3 install \
  keras_preprocessing==1.1.0 --no-deps
sudo pip3 install h5py==2.9.0
```

```
sudo pip3 install pybind11
pip3 install -U --user six wheel mock
wget "https://raw.githubusercontent.com/PINTO0309/Tensorflow-
bin/master/tensorflow-2.4.0-cp37-none-
linux_armv7l_download.sh"
chmod +x ./tensorflow-2.4.0-cp37-none-
linux_armv7l_download.sh
./tensorflow-2.4.0-cp37-none-linux_armv7l_download.sh
sudo pip3 uninstall tensorflow
sudo -H pip3 install tensorflow-2.4.0-cp37-none-
linux_armv7l.whl
```

Die letzten fünf Befehle laden ein Downloader-Skript, machen es ausführbar, führen es aus, deinstallieren eventuell vorhandene alte Versionen und installieren das heruntergeladene Python-Binärpaket (ein "Wheel", Endung *.whl*

Tensorflow ist leider etwas wählerisch bei den Versionen seiner Abhängigkeiten. Sollte ein Versionssprung dazu führen, dass die Befehle bei Ihnen nicht so wie hier gedruckt funktionieren, müssen Sie einen Blick in die Anleitung im Repository werfen.

Datensatz herunterladen

Das Modul *tensorflow_datasets* bietet zwar einfachen Zugriff auf viele populäre Datensätze - auch Quick Draw, es bindet aber immer den vollen Datensatz ein, der den Speicher der Raspis sprengt. Glücklicherweise bietet Google die Zeichnungen für Obst und Gemüse auch als einzelne Dateien zum Download an, die Sie mit wenigen Zeilen Python- und Numpy-Code zum Datensatz aufbereiten (siehe *desktop_load_quick_draw.py* im Git-Repository, zu finden über ct.de/ykkz).

Das Skript definiert zunächst das Obst und Gemüse als Liste:

```
items = ['apple',
  # ...
  'watermelon']
```

Die Daten lädt es von *storage.googleapis.com* mit *urllib.request.urlretrieve()* in den Ordner *DATA_PATH ='./data/'* . Dort landet dann eine Datei pro Gemüse, die das Skript zu einem Trainings- und einem Validierungs-Datensatz umpackt. Im Prinzip liegen die Daten bereits als binär kodierte Tabellen vor. Das Skript packt alle Bilder in eine lange Liste und erstellt passend dazu eine Liste mit der Gemüsesorte als Label, also der Information, die das trainierte Netz ausgeben soll.

Die Liste mischt das Skript gleich noch durch, damit beim Training nicht alle Beispiele zu einer Gemüsesorte hintereinander stehen. Neuronale Netze lernen üblicherweise am besten, wenn Sie in kurzer Folge mit der gesamten Bandbreite an Beispielen konfrontiert werden.

Zuletzt schneidet das Skript 10 Prozent der Daten für die Validierung ab. Diese Daten sieht das Netz beim Training nie, weshalb es sie nicht auswendig lernen kann. Deswegen können Sie nach einem Test mit dem Validierungs-Datensatz abschätzen, ob das Training auch mit unbekannten Daten funktionieren wird.

Am Ende legt das Skript vier Dateien im Ordner *data* ab, die Sie mit *scp* auf alle vier Raspis kopieren:

```
scp -r data pi@raspi1:/home/pi/
```

Verteilte Trainingsdaten

Um datenparallel zu trainieren, muss jeder beteiligte Rechner einen Teil der Daten bekommen. Bei der Verteilung hilft die Klasse *Tensorflow.data.Dataset* . Sie kapselt die zuvor geladenen Numpy-Arrays so, dass die *fit()* -Methode eines mit Keras definierten

Model die Daten effizient durch das neuronale Netz schleusen kann. Damit der Trainingsalgorithmus die Beispiele dabei mehr als einmal nutzen kann, hängt *.repeat()* die Daten zu einer beliebig langen Liste hintereinander. Die Methode *.batch()* liefert einen Happen an Daten, mit denen das neuronale Netz gleichzeitig alle Neuronen simuliert, mit den gewünschten Ausgaben vergleicht und die Fehler zurückverfolgt:

```
x_train, y_train = load_data()
x_train = x_train / np.float32(255)
y_train = y_train.astype(np.int32)
train_dataset = tf.data.Dataset.\
  from_tensor_slices((x_train,
  y_train)).repeat().batch(batch_size)
```

Die erste Zeile lädt die Daten als Numpy-Arrays, die beiden darauffolgenden Zeilen normieren die Daten auf den Wertebereich 0 bis 1 und setzen die Datentypen. Danach erzeugt die Factory-Methode *tf.data.Dataset.from_tensor_slices()* ein *Dataset* -Objekt und *.repeat()* und *.batch()* stellen für dieses ein, dass es beliebig oft häppchenweise Daten liefert.

Seit TensorFlow 2.0 bereitet man Datensätze immer in dieser Form auf. Damit TensorFlow die Daten automatisch an die Raspis verteilt, muss man zusätzlich Optionen setzen. Ein *Dataset* hat dafür ein *Options* -Objekt, dessen *auto_shard_policy* auf *DATA* stehen sollte. Dann nämlich zerteilt TensorFlow jeden Batch in gleich große Bruchstücke (Shards) und überträgt sie an die beteiligten Rechner. Die folgenden drei Zeilen stellen diese Option ein:

```
options = tf.data.Options()
options.experimental_distribute.\
  auto_shard_policy = tf.data.\
  experimental.AutoShardPolicy.DATA
train_dataset = train_dataset.\
  with_options(options)
```

Die Form des Netzwerks

Wie ein mit Keras definiertes neuronales Netz aussieht, definiert ein *Model* . Da die meisten neuronalen Netze die Neuronen einfach schichtweise übereinanderstapeln und die Schichten der Reihe nach berechnen, hilft die Klasse *Sequential* , der man einfach eine Liste mit Neuronenschichten übergibt und die die Verbindungen zwischen denen automatisch verdrahtet:

```
model = tf.keras.Sequential([
  tf.keras.layers.Flatten(
              input_shape=(28, 28)),
  tf.keras.layers.Dense(
              256, activation='relu'),
  tf.keras.layers.Dropout(0.2),
  tf.keras.layers.Dense(
              number_of_items,
              activation='softmax')
])
```

Die erste Schicht sollte mit *input_shape* festlegen, welche Form die Eingaben haben sollten. Bei QuickDraw sind das Graustufenbilder mit 28 × 28 Pixeln, was auf eine 28×28-Matrix hinausläuft. Die Schicht *Flatten* macht daraus einen eindimensionalen Vektor mit 784 Zahlen.

Eine *Dense* -Schicht verbindet jede Eingabe mit jeder Ausgabe. Jede Verbindung

hat ein eigenes trainierbares Gewicht, weshalb die zweite Schicht 200.960 Parameter beiträgt.

Die dritte Schicht (*Dropout*) verwirft während des Trainings 20 Prozent der Daten (sie setzt die Aktivierung von 20 Prozent der Neuronen auf 0). Das hilft, das Netzwerk zu zwingen, übergreifende Konzepte zu lernen und nicht nur Trainingsdaten 1:1 zu reproduzieren (Overfitting).

Die *Dense* -Schicht Nummer vier ist schon die Ausgabeschicht. Sie hat ein Neuron für jedes Obst oder Gemüse, das das Netzwerk erkennen kann (3341 Parameter). Eine nicht lineare Ausgabefunktion wird hier nicht angewendet. Stattdessen normiert *activation='softmax'* die Ausgaben so, dass sie sich zu 1 addieren. Damit sieht die Ausgabe wie eine Wahrscheinlichkeitsverteilung aus und man kann direkt ablesen, wie wahrscheinlich es das Netzwerk findet, ein bestimmtes Gemüse vor sich zu haben.

Um so ein *Model* zu trainieren, muss man Keras noch einen Trainingsalgorithmus mitgeben. Das geht ganz einfach, indem man der Funktion *compile()* einen *optimizer* überreicht. Damit der arbeiten kann, braucht er noch eine *loss* -Funktion, die jeweils berechnet, wie falsch das Netzwerk gemessen an den Daten lag. Standardmäßig berechnet Keras beim Training nur *loss* . Um als Mensch den Fortschritt zu beurteilen, ist aber die Metrik *accuracy* ein besserer Wert. Die berechnet nämlich, wie oft das Netzwerk richtig rät, die höchste Wahrscheinlichkeit also zum richtigen Gemüse gehört. Im Code sieht der gesamte Lernalgorithmus folgendermaßen aus:

```
model.compile(loss=
    'sparse_categorical_crossentropy',
 optimizer=tf.keras.optimizers.Adam(),
 metrics=['accuracy'])
```

Cluster einrichten

Das *Model* (siehe Kasten) wird beim datenparallelen Lernen einfach unverändert an die beteiligten Rechner verteilt. Deswegen fehlt jetzt nur noch die Angabe, wer alles mitrechnet. Das stellt die Umgebungsvariable *TF_CONFIG* ein. In diesem JSON-Objekt sollte es ein *"cluster"* geben, das eine Liste von *"worker"* definiert (Hostname und ein offener Port) und in *"task"* angibt, welche Rolle der einzelne Rechner spielt. In der Praxis definiert sich einfach jeder Raspi als *"worker"* und stellt *"index"* auf die Stelle, an der der eigene Hostname in der Liste der *"worker"* vorkommt. Für *raspi1* sieht das wie folgt aus:

```
TF_CONFIG = {"cluster":
  {"worker": ["raspi1:54321",
              "raspi2:54322",
              "raspi3:54323",
              "raspi4:54324"]},
   "task": {"index": 0,
            "type": "worker"}}
```

Die drei anderen Raspis nutzen dann *"index"* 1 bis 3. Die folgenden Zeilen suchen den richtigen Index anhand des Hostnamens heraus und tragen ihn in Zeile 2 in das *TF_CONFIG* -Dictionary ein:

```
hostname = os.uname().nodename
tf_config['task']['index'] = \
  WORKERS.index(hostname)
```

Aus dem Python-Dictionary macht die *json* -Bibliothek den passenden String und *os.environ()* setzt die Umgebungsvariable:

```
os.environ['TF_CONFIG'] = json.dumps(
                            tf_config)
```

Verteilt trainiert

Damit TensorFlow mit der *MultiWorkerMirroredStrategy* trainiert, muss das Trainingsskript auf den Raspis diese nur erzeugen und *model.fit()* in deren *scope()* ausführen:

```
strategy = tf.distribute.\
  MultiWorkerMirroredStrategy()
with strategy.scope():
  model = build_and_compile_model(13)
  model.fit(dataset, epochs=32,
   steps_per_epoch=256,
   validation_data=validation_dataset)
```

Dieser Code muss auf jedem der vier Raspis laufen. Das geht prinzipiell mit Paramiko (siehe *train_on_workers.py* im Git-Repository), das Einsammeln der Meldungen von TensorFlow läuft dann aber verzögert ab. Deswegen ist es bei nur vier Raspis sinnvoller, vier Konsolenfenster zu öffnen, sich in jedem per SSH zu einem der Raspis zu verbinden und das Skript dort mit *python worker.py* zu starten. In den Terminals funktionieren die Fortschrittsbalken dann auch wie gewohnt und man kann live verfolgen, wie die Raspis synchron Batch für Batch verarbeiten.

Per WLAN ausgebremst

Die vier Raspis brauchten zum Lernen mit den angegebenen Parametern etwa eine halbe Stunde. Die Treffergenauigkeit (Accuracy) lag bei den Testdaten bei 73 Prozent - weit besser als Raten.

Setzt man die Umgebungsvariable *TF_CONFIG* nicht, trainiert TensorFlow nur auf einem Raspi statt auf allen vier. Die Fortschrittsbalken füllen sich dann viel schneller und nach zwei Minuten ist das Training komplett fertig. Das zeigt, dass mehr Rechner nicht automatisch besser sind: Die vier Raspis kommunizieren mit hoher Latenz über WLAN, um nach jedem Batch alle Parameter abzugleichen. In dieser Zeit warten die Prozessoren der Raspis, bis die Netzwerkpakete angekommen sind, weshalb die Raspis nie über 20 Prozent Auslastung erreichen. Deswegen geht das Training auf einem Raspi erheblich schneller als auf vier davon. Mit Ethernet-Kabeln vernetzte Raspis arbeiten etwas schneller zusammen, was sie aber trotzdem nicht zu empfehlenswerter Hardware für mehr als KI-Fingerübungen macht.

Das Raspi-Cluster ist also geeignet, Einstellungen für verteiltes Lernen mit TensorFlow kostengünstig auszuprobieren. Wenn Sie mit verteiltem Lernen wirklich Zeit sparen wollen, sollten Sie jedoch Cluster-Knoten anmieten, die mit einem latenzarmen und breitbandigen Interconnect wie Infiniband vernetzt sind. Die Prozessoren in solchen Knoten beherrschen auch lange Vektorbefehle und rechnen um Größenordnungen schneller als ein Raspi.

Literatur

1. Pina Merkert, "Hallo Welt" der KI, Mit der Python-Bibliothek Keras Deep-Learning-Algorithmen selbst programmieren, c't 21/2019, S. 32
2. Andrea Trinkwalder, Netzgespinste, Die Mathematik neuronaler Netze: einfache Mechanismen, komplexe Konstruktion, c't 6/2016, S. 130
3. Ronald Eikenberg, Raspi-Schnellstart, Raspberry Pi superschnell einrichten, c't 11/2021, S. 132

Code auf GitHub: ct.de/ykkz

Gerhard Völkl

Quelle:	c't Heft 12/2021 S. 130-135
ISSN:	0724-8679
Ressort:	Wissen
Rubrik:	TensorFlow auf 4 Raspis
Dokumentnummer:	2110908020402798026