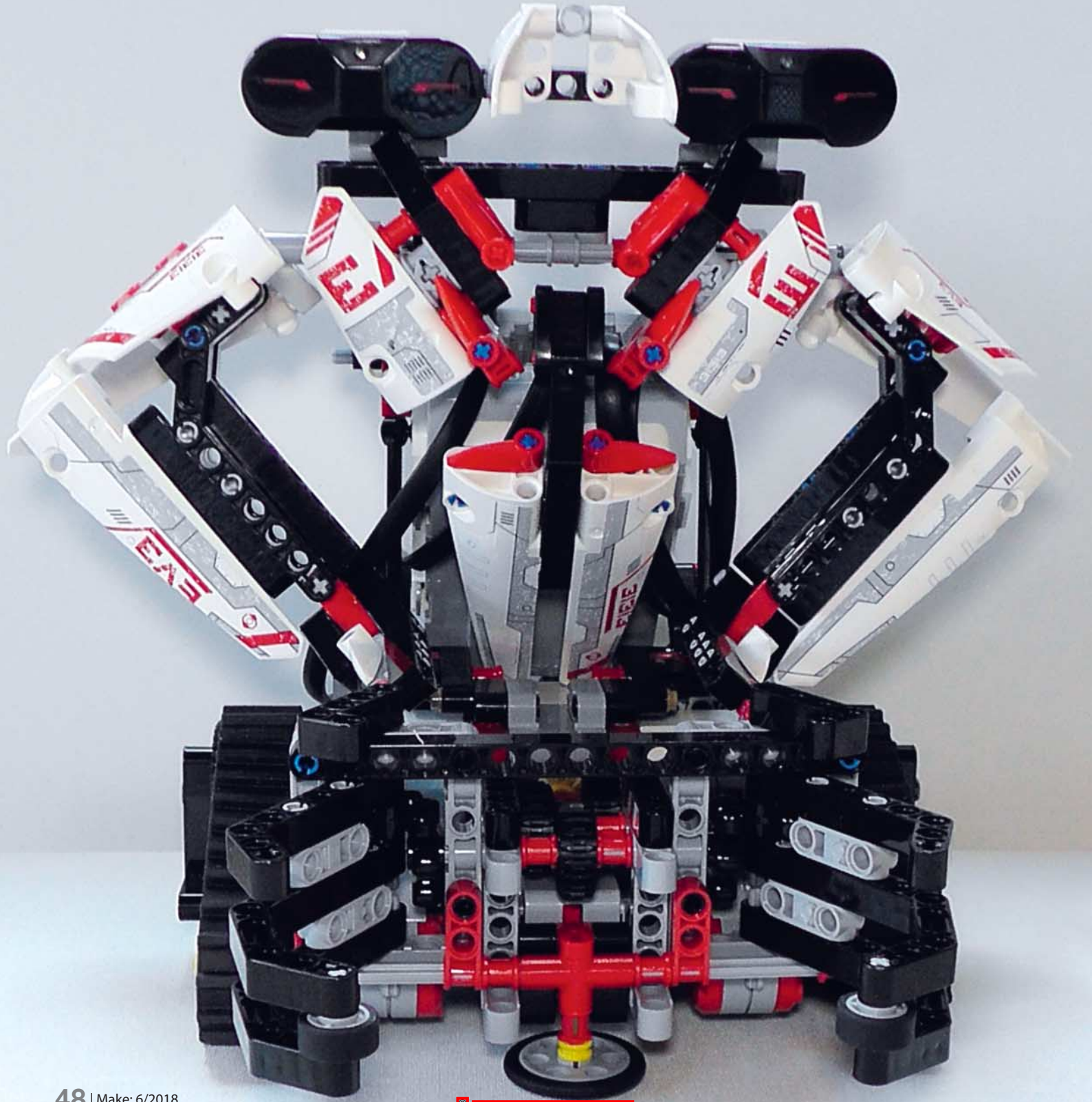
KI für Lego-Roboter

Künstliche Intelligenz und maschinelles Lernen kommen immer mehr in der Praxis zum Einsatz. Davon motiviert bauen wir einen Lego-Mindstorms-EV3-Roboter, den ein Benutzer mit sieben Gesten steuern kann. Wir realisieren die Gestenerkennung mit einem neuronales Netz, trainieren es auf dem PC und bauen es in den Roboter ein.

von Detlef Heinze



Menschen und Roboter arbeiten immer häufiger zusammen, und es gibt Situationen, in denen sie sich ohne Sprache koordinieren müssen. Gründe dafür können zum Beispiel laute Umgebungen sein oder die Anforderung, in Sekundenbruchteilen einem Roboter einen Auftrag zu erteilen, ohne die relativ langsame menschliche Sprache verwenden zu können. Auch wenn viele Menschen und Roboter in einem Raum gleichzeitig arbeiten, können Gesten eine nützliche und für Menschen einfache Möglichkeit sein, Roboter zu steuern. Ebenso lösen Autofahrer bereits heute in einigen Fahrzeugen ausgewählte Funktionen mit Gesten aus.

Als Gesten werden hier Zeichen mit einer Hand verstanden, die in unmittelbarer Nähe zum Sensor eines Roboters durch einen menschlichen Nutzer ausgeführt werden und eine Aktion seitens des Roboters auslösen. Eine solche Aktion kann nur eine kleine Bewegung des Roboters, aber auch eine ganze Aufgabenerledigung sein.

In diesem Artikel stellen wir einen Lego-Sensor vor, mit dem man Lego-Roboter mittels Gesten steuern kann. Dabei wenden wir die Technik des maschinellen Lernens zur Wiedererkennung der sieben Gesten an, die von zwei Infrarotsensoren gescannt werden. Bei dieser Klassifikationsanwendung kommt überwachtes Lernen zum Einsatz.

Dieser Artikel besteht aus zwei Teilen. Der erste Teil erklärt die Grundlagen einfacher neuronaler Netze sowie die notwendigen Begriffe, um ein neuronales Netz realisieren zu können. Im zweiten Teil beschreiben wir dann, wie der Roboter realisiert wird. Er trägt den Namen „Gestur3Bot“.

Kurzinfo

- **Neuronale Netze verstehen**
- **Lego-Roboter bauen**
- **Neuronales Netz realisieren, trainieren und anwenden**

Checkliste

Zeitaufwand: ab 6 Stunden

Programmieren: Python

Alles zum Artikel im Web unter *make-magazin.de/xt34*

Hardware

- **Lego-Mindstorms-EV3-Baukasten**
- **Ein zusätzlicher EV3-Infrarot-Sensor**
- **PC mit Windows 7 (64 Bit)** oder höher

Software (kostenlos)

- **Lego-Mindstorms-EV3-Programmiersoftware**
- **Python 3.6.3 (64bit),** mit der aktuellen Version 3.7.1 funktioniert Tensorflow nicht mehr
- **Library TensorFlow 1.8** oder höher
- **Library Keras 2.1.3** oder höher
- **Library Fileformat h5py**
- **Ein beliebiger Texteditor,** z. B. Notepad++
- **Das Github-Repository** zum Projekt

Grundlagen

Neuronale Netze orientieren sich am Vorbild der Natur. Sieht und erkennt ein Mensch beispielsweise ein Auto, so entsteht auf der Netzhaut des Auges zunächst ein Reizmuster, das an die verbundenen Nervenschichten im Gehirn darunter weitergeleitet und in immer tieferen Schichten von Nerven ausgewertet wird. Am Ende erkennen wir ein Auto. Die Fähigkeit, ein Auto als Auto zu erkennen, ist uns aber nicht angeboren. Das lernen wir anhand von Beispielen in der Kindheit.

Dieses Prinzip des Erkennens durch Lernen von Beispielen nutzen auch neuronale Netze. Ihnen ist die Fähigkeit zu lernen gegeben, ohne dass zunächst festgelegt ist, was gelernt wird. Dadurch sind sie, wie unser Gehirn, sehr flexibel für unterschiedliche Anwendungen einsetzbar. Ein Großteil des Aufwandes herkömmlicher Programmierung verschiebt sich bei neuronalen Netzen daher ins Lernen: dem Beschaffen von guten Beispielen, den Trainings- und Testdaten sowie die eigentliche Trainingsphase. Wir programmieren also viel weniger und müssen uns dafür umso mehr mit Daten und dem Training beschäftigen.

Betrachten wir ein neuronales Netz als Blackbox von außen, so besitzt es eine Schicht von Neuronen für die Eingabe und eine Schicht für die Ausgabe. Die Eingabeschicht nimmt in unserem Fall des Gestur3Bot die von den beiden Infrarotsensoren gemessenen Abstandswerte einer sich bewegenden Faust auf, dazu später mehr.

Beide Sensoren ermitteln zusammen 32 Werte, die als Merkmalsvektor x bezeichnet werden. Für jede einfache KI-Anwendung muss man die Anzahl und Werte-Typen des Merkmalsvektors festlegen, bevor man ein neuronales Netz dafür definiert. Die Größe des Vektors und seine Wertebereiche müssen für die Trainingsphasen und für die Anwendung konstant bleiben. Es dürfen sich nur die Inhalte ändern. Auch alle Trainings- und Testdaten müssen aus der gleichen Anzahl von Werten bestehen. Im vorliegenden Fall besteht der Vektor aus 32 Werten zwischen 0 und 75.

Das neuronale Netz steht für eine mathematische Funktion $y = f(x)$, die aus dem Merkmalsvektor x den hoffentlich richtigen Ausgabevektor y berechnet. Allgemein wird diese Berechnung auch Inferencing genannt.

Als Ausgabe liefert das neuronale Netz den Vektor y, also das Ergebnis des neuronalen Netzes. Bei der Erkennung von Gesten handelt es sich um eine sogenannte Klassifikation, bei der jedem Merkmalsvektor x eine der möglichen sieben Klassen (Gesten) zugeordnet wird. Aus diesem Grund besteht der Ausgabevektor y aus 7 Werten, die jeweils die Wahrscheinlichkeit 0 bis 1 für eine erkannte Geste enthalten. Wert 1 des Ausgabevektors ist der ersten Geste zugeordnet, Wert 7 der siebten Geste. Auch wenn man nur eine Geste zeigt, sind vermutlich mehrere der sieben Werte größer als Null. Der Wert mit der höchsten Wahrscheinlichkeit ist dann vermutlich die gezeigte Geste.

Die Trainings- und Testdaten bestehen aus einer größeren Zahl von Merkmalsvektoren. Dabei ist jedem Vektor x die richtige Klasse y – also die Geste – bereits zugeordnet, sodass in der Trainingsphase das neuronale Netz die richtigen Klassifizierungen lernen kann. Man nennt diese Art des Trainings überwachtes Training, da die richtigen Klassifizierungen vorgegeben sind. Im Vorfeld des Trainings muss man diese Daten zunächst erstellen oder beschaffen, was je nach Anwendung einen großen Aufwand erzeugen kann.

Praxis

Nun stellt sich die Frage, wie ein neuronales Netz im Innern aufgebaut ist und wie es rechnet. Neben der Ein- und Ausgabeschicht kann das neuronale Netz keine, eine oder viele sogenannte versteckte Schichten von Neuronen besitzen. Sie werden eine nach der anderen aufgereiht und zwischen die

Ein- und Ausgabeschicht eingesetzt. So entsteht eine sequentielle Folge von Neuronenschichten. Das neuronale Netz des Gestur3Bot besitzt eine einzige versteckte Schicht mit 20 Neuronen (1).

Neuronen sind jedoch keine isolierten Objekte. Die Fähigkeit zum Lernen entsteht durch ihre Verbindungen untereinander. Bei einfachen neuronalen Netzen verbindet man alle Neuronen der Vorgängerschicht mit allen Neuronen der Nachfolgerschicht. So ergeben sich für den Gesture3Bot bei einem Merkmalsvektor mit 32 Werten und einer versteckten Schicht mit 20 Neuronen schon 32 • 20 = 640 Verbindungen nur zwischen diesen beiden Schichten. Hinzu kommen noch 20 • 7 = 140 Verbindungen zwischen der versteckten Schicht und der Ausgabeschicht. In Summe hat unser „kleines Gehirn" 640 + 140 = 780 Verbindungen zwischen den Neuronen.

Der Wert eines jeden einzelnen Neurons berechnet sich in Abhängigkeit von der Anzahl seiner Verbindungen n (bei uns 32 oder 20) zur Vorgängerschicht auf folgende Weise:

$$\sum_{i=1}^{n} w_i x_i + b$$

wobei x_i der Wert des i-ten Vorgängerneurons und w_i die Gewichtung (engl. weight) der i-ten Verbindung sind. Die Gewichtungen w_i und der zusätzliche Wert b (engl. bias) sind zunächst unbekannte Werte. Sie sind es, die beim Training des neuronalen Netzes durch einen Lernalgorithmus mit Hilfe der Trainingsdaten festgelegt werden müssen.

Eine entscheidend wichtige Eigenschaft von biologischen Neuronen fehlt uns aber noch. Natürliche Neuronen erzeugen nur ein Ausgangssignal – man sagt auch, sie „feuern" –, wenn eine bestimmte Reizschwelle überschritten wird. Um ein derartiges Verhalten zu simulieren, verwenden wir eine sogenannte „Aktivierungsfunktion". Eine einfache und effiziente Funktion ist die Rectified Linear Unit (ReLU), die für negative Werte 0 und für positive Werte den positiven Wert unverändert zurückgibt:

$$relu\,(v) = max\,(0,v)$$

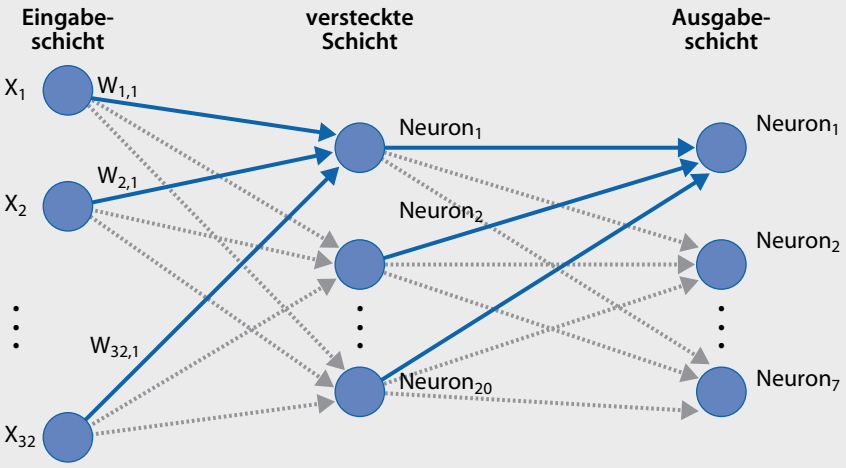
Somit erhalten wir mit der obigen Summenformel die Berechnung für ein Neuron in einer versteckten Schicht. Der Wert wird auch als „logit" bezeichnet.

$$logit_{neuron} = max\left(\sum_{i=1}^{n} w_i x_i + b, 0\right)$$

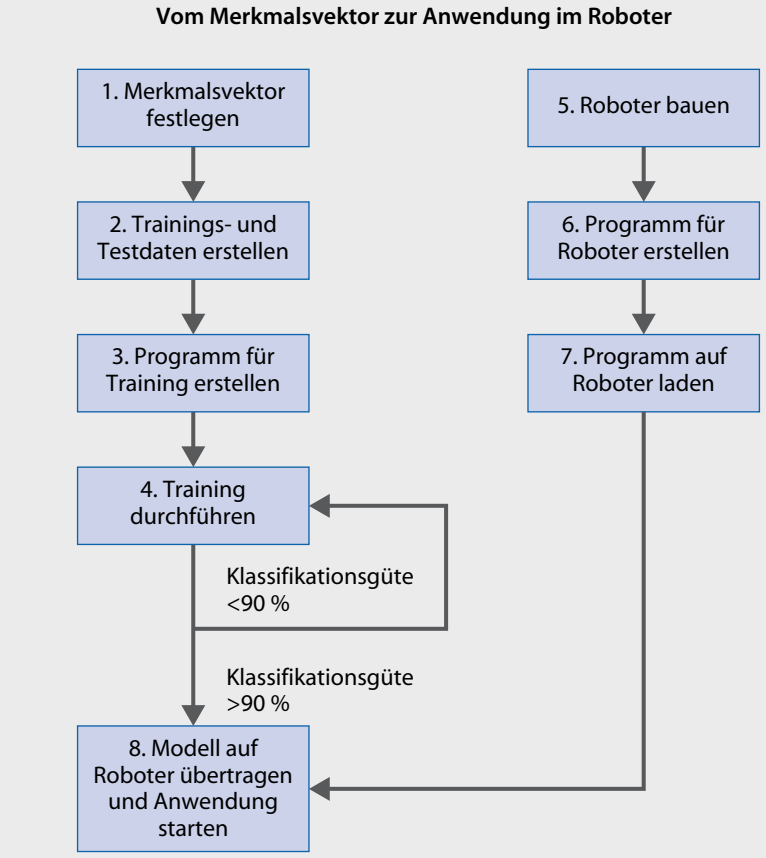
Bei der Berechnung der Neuronen in der Ausgabeschicht lässt man die ReLU-Funktion weg, da wir das Ergebnis unverändert für die Auswertung der erkannte Klasse benötigen.

Ein ganzes neuronales Netz wird streng von links (Eingabeschicht) nach rechts zur nächsten versteckten Schicht berechnet. Ist

Übersicht über das neuronale Netz des Gestur3Bot (1)



Realisierungsschritte im Überblick (2)



die berechnet, kommt die nächste versteckte Schicht an die Reihe usw., bis die Ausgabeschicht mit dem Ergebnis erreicht ist. Hilfreich ist, dass eine ganze Neuronenschicht mit einer Matrizenmultiplikation und einer Matrizenaddition und anschließender Anwendung von ReLU gemäß der obigen Gleichung realisiert werden kann. Dies sehen wir später bei der Implementierung des Inferencing im Roboter.

Alle Gewichtungen und Bias-Werte des gesamten neuronalen Netzes zusammen werden als Modell bezeichnet. Es ist das Ergebnis des Trainings und besteht in unserem Fall aus 807 Werten (780 Gewichtungen + 20 + 7 Bias-Werte). Das Modell müssen wir auf den Roboter übertragen.

Die Qualität des Modells kann während und nach dem Training gemessen werden. Dazu verwenden wir die Testdaten, die nicht am Training teilnehmen. So können wir feststellen, wie gut bislang unbekannte Daten klassifiziert werden. Das Maß dafür ist die Klassifikationsgüte, die den Prozentsatz der richtig klassifizierten Testdaten wiedergibt. Eine Klassifikationsgüte von über 90 % ist gut. Eine Klassifikationsgüte in der Nähe von 100 % erreichen auch professionelle Anwendungen mit mehreren Layern kaum.

Zu Beginn eines jeden Trainings initialisiert das Trainingsprogramm ein Modell mit Zufallszahlen. Darum fällt auch bei jedem Training die Klassifikationsgüte unterschiedlich aus. Ein Vorteil dieses Vorgehens ist, dass gelegentlich ein Training mit besonders hoher Klassifikationsgüte abschließt. Dieses Modell verwenden wir für den Einsatz im Roboter.

Zugriff auf GitHub

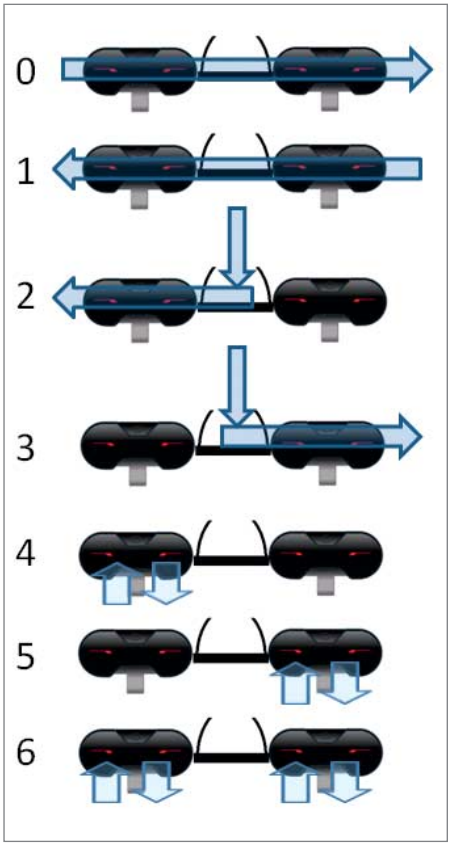
Unter den Kurzinfos am Anfang des Artikels finden sie einen Link, unter dem Sie weitere Links und Infos finden, darunter auch den Link zur GitHub-Seite dieses Projekts. Dort klicken Sie den grünen Button „Clone or download“ und anschließend den Button „Download ZIP“. Auf dem PC entpacken Sie das ZIP-Archiv in einen Ordner ihrer Wahl.

Mit Gesten steuern

Nun wollen wir den Roboter realisieren. 2 zeigt alle Bauabschnitte und ihre Reihenfolge. Zum Nachbau müssen wir nur die Schritte 5, 7 und 8 ausführen. Die Ergebnisse der anderen Schritte sind im GitHub-Repository abgelegt.

Der Gestur3Bot ist eine leichte Modifikation des Gripp3r-Roboters aus dem Mindstorms-EV3-Baukasten. Die Lego-Bauanleitung des Gripp3r kann bei Lego heruntergeladen werden. Die Adresse und die Beschreibung der Modifikationen inklusive Fotos befinden sich in einer Readme-Datei des GitHub-Repository.

Die 7 Gesten führt der Benutzer mit der Hand aus 3. Genauer gesagt führt er sie mit der Faust aus, da sie vorne eine größere Rückstrahlfläche für das IR-Licht besitzt. Der Gestur3bot hat eine „Nase“ zwischen den IR-Sensoren, die verhindern soll, dass von der Faust des Benutzers zurückgestrahltes IR-Licht des einen Sensors auf den Empfänger des jeweils anderen Sensors fällt. Die Gesten muss man mit mittlerer Geschwindigkeit ausführen, das Video im GitHub-Repository zeigt, wie.



3 Die 7 Gesten des Gestur3Bot

Die 7 Gesten sind von 0 bis 6 durchnummeriert. Sobald der GestureBot eine Geste erkannt hat, sagt er die Nummer über den Lautsprecher an.

```
1.0 0.0 0.0 0.0 0.0 0.0 0.0
1.0 0.0 0.0 0.0 0.0 0.0 0.0
1.0 0.0 0.0 0.0 0.0 0.0 0.0
1.0 0.0 0.0 0.0 0.0 0.0 0.0
1.0 0.0 0.0 0.0 0.0 0.0 0.0
1.0 0.0 0.0 0.0 0.0 0.0 0.0
1.0 0.0 0.0 0.0 0.0 0.0 0.0
1.0 0.0 0.0 0.0 0.0 0.0 0.0
1.0 0.0 0.0 0.0 0.0 0.0 0.0
1.0 0.0 0.0 0.0 0.0 0.0 0.0
1.0 0.0 0.0 0.0 0.0 0.0 0.0
1.0 0.0 0.0 0.0 0.0 0.0 0.0
1.0 0.0 0.0 0.0 0.0 0.0 0.0
1.0 0.0 0.0 0.0 0.0 0.0 0.0
0.0 1.0 0.0 0.0 0.0 0.0 0.0
0.0 1.0 0.0 0.0 0.0 0.0 0.0
0.0 1.0 0.0 0.0 0.0 0.0 0.0
0.0 1.0 0.0 0.0 0.0 0.0 0.0
0.0 1.0 0.0 0.0 0.0 0.0 0.0
0.0 1.0 0.0 0.0 0.0 0.0 0.0
0.0 1.0 0.0 0.0 0.0 0.0 0.0
0.0 1.0 0.0 0.0 0.0 0.0 0.0
0.0 1.0 0.0 0.0 0.0 0.0 0.0
0.0 1.0 0.0 0.0 0.0 0.0 0.0
```

```
15 11 13 26 26 40 44 42 45 48 48 49 51 53 54 54 42 42 41 36 14 8 8 5 9 9 36 28 28 29 30 31
18 13 12 12 12 12 14 19 19 28 40 43 42 44 45 45 43 43 43 42 42 41 41 39 36 18 10 8 5 5 5 9
22 16 16 12 10 11 12 18 18 27 40 40 43 43 44 45 44 44 44 43 43 42 40 40 36 20 12 9 8 10 10 16
20 20 14 11 9 9 10 10 15 27 42 45 47 47 50 50 43 43 43 43 42 42 42 40 36 18 9 5 4 4 4 9
20 15 15 12 11 11 14 18 18 26 38 38 44 45 47 47 43 43 43 42 42 42 39 36 22 15 15 10 8 6 6 6
17 12 12 12 17 32 32 44 44 46 47 48 51 51 51 51 43 43 42 39 39 36 13 8 7 7 7 10 14 14 36 28
20 13 10 10 10 10 13 19 30 30 42 42 44 45 47 47 43 43 43 42 42 41 36 36 24 14 8 5 5 5 6 12
20 13 11 11 11 12 16 24 35 35 44 43 45 46 47 47 43 43 43 42 42 39 36 21 21 13 8 6 5 6 10 10
21 21 15 11 11 11 12 12 16 25 39 39 45 45 47 48 47 43 43 43 42 42 42 40 36 22 13 8 5 5 6 10
15 12 9 9 7 7 7 8 10 16 16 32 32 46 49 54 43 43 43 43 42 42 42 41 38 36 16 16 8 5 5 8
17 10 10 8 7 7 11 11 11 37 42 45 46 47 47 50 43 43 43 43 42 42 39 36 14 8 8 5 5 6 8 15
16 9 7 7 8 15 15 40 43 46 46 47 47 48 49 49 43 43 43 42 39 36 11 11 8 5 5 9 12 12 12 36
15 9 8 8 7 8 12 12 36 43 43 43 45 46 47 48 43 43 43 43 42 41 36 20 20 10 8 5 6 10 10 10
21 14 10 8 8 8 9 13 13 36 36 43 43 45 46 47 45 44 43 43 43 42 42 38 36 16 16 9 8 7 9 16
19 13 10 8 8 8 8 12 12 36 36 41 41 44 45 46 43 43 43 43 43 43 42 39 36 17 17 10 8 5 5 7
17 17 11 9 7 8 10 10 16 28 39 44 43 43 45 49 43 43 43 43 43 42 41 36 21 21 11 8 5 5 6 6
18 18 12 9 7 7 9 13 13 13 36 43 42 44 44 46 43 43 43 43 43 43 42 40 36 17 17 9 6 5 5 6
21 14 14 10 9 7 8 9 9 13 13 36 39 43 43 42 43 43 43 43 43 43 43 42 41 36 21 21 12 8 5 5
21 14 9 7 7 7 7 11 11 36 36 41 41 44 46 47 43 43 43 43 43 43 42 40 40 36 13 8 6 6 10 10
23 16 16 11 9 7 8 9 9 13 13 37 42 44 46 46 44 44 43 43 43 43 42 42 41 36 22 11 8 5 5 7
18 14 14 12 12 15 21 21 36 36 46 47 48 50 50 49 43 43 43 43 42 41 36 17 17 9 7 8 16 16 16 36
19 15 14 14 16 16 21 31 44 46 48 48 52 61 71 71 43 43 43 42 42 38 38 36 16 11 9 9 9 12 12 36
21 14 11 11 10 11 15 23 38 38 38 45 47 52 58 57 43 43 43 43 43 42 42 36 19 9 8 8 7 10 16 16
21 15 13 13 13 17 25 42 44 44 46 48 48 50 52 52 43 43 43 43 43 42 42 38 36 11 8 7 10 10 15 15 36
16 13 13 16 16 22 34 34 45 47 47 48 49 49 49 49 43 43 43 42 41 36 17 11 11 9 9 11 16 16 16 36
12 10 12 12 19 33 39 44 43 45 45 47 48 48 49 49 43 43 42 41 36 12 8 5 5 6 13 13 36 28 26 26
21 21 14 10 10 13 18 18 30 40 44 43 45 47 47 49 43 43 43 43 43 42 38 38 36 11 8 7 8 13 13 13
12 10 10 11 13 17 25 35 35 41 42 44 46 51 63 63 43 43 43 43 43 45 40 36 18 11 11 8 7 7 8 11 16
14 10 10 10 14 24 40 40 45 45 48 58 61 62 62 61 43 43 43 42 37 37 36 9 7 9 15 15 15 36 29 30
19 13 13 10 10 13 19 32 32 42 42 44 46 50 50 53 43 43 43 43 43 42 41 36 13 9 7 7 9 12 12 36
26 20 15 15 12 11 11 11 14 14 18 27 37 43 43 43 43 43 43 43 43 43 43 43 42 41 36 20 12 12 8 7
15 15 10 10 13 20 34 34 44 43 46 48 50 50 51 51 43 43 43 43 41 41 36 14 9 8 11 11 11 36 29 31
```

Auszüge aus der Folge vieler Merkmalsvektoren (32 Werte) und den dazugehörigen Klassen (Labels) mit 7 Werten (links)

Gesten definieren

Der Merkmalsvektor einer Geste besteht aus 32 Sensorwerten. Davon kommen jeweils 16 Werte von jedem Sensor. Jede einzelne Messung ist ein Abstandswert zur Faust. Beide Sensoren messen gleichzeitig. So wird der Weg der Faust im Merkmalsvektor aufgezeichnet.

Für die Geste 0, die der Benutzer vor dem Roboter mit einer Bewegung der Faust von links nach rechts ausführt, sieht ein Merkmalsvektor beispielsweise folgendermaßen aus. Die Werte der erste Zeile kommen vom linken Sensor (vom Benutzer aus gesehen) und die der zweiten Zeile vom rechten Sensor. Die Zeit läuft von links nach rechts. Die jeweils übereinander stehenden Werte erfassen die Sensoren gleichzeitig.

```
17 12 12 10 11 20 42 54 54 55 55 56
55 55 55 55
42 39 36 34 32 24 12 12  8  4  8  8
39 39 44 45
```

Der Weg der Faust von links nach rechts ist nachvollziehbar: Erst sind die Abstandswerte des linken Sensors klein und später die des rechten Sensors. Da ein Benutzer jedoch jede Ausführung einer Geste leicht unterschiedlich in Geschwindigkeit und Abstand zum Sensor ausführt, variieren die gemessenen Werte merklich. Gerade diese Variation ist es, die mit einem normalen Programm schwierig zu erfassen ist, für deren Verarbeitung ein neuronales Netz jedoch prädestiniert ist, um den Merkmalsvektor trotzdem sehr häufig der richtigen Klasse zuordnen zu können.

Nun müssen wir uns im zweiten Schritt von Bild 2 um die Trainings- und Testdaten kümmern. Im GitHub-Repository (siehe Link) ist ein kompletter Satz von Trainings- und Testdateien im Unterordner „Data“ enthalten. Pro Klasse enthält die Trainingsdatei 980 Merkmalsvektoren, 140 pro Klasse respektive Geste (xTrain_Gesture0-6_980-32.csv). Die Testdatei ist kleiner und besteht aus 420 Merkmalsvektoren, 60 pro Klasse (xTest_Gesture0-6_420-32.csv). Da wir überwachtes Training durchführen, benötigen wir noch die Klassenangabe zu den Merkmalsvektoren. Sie sind in den Dateien yTrain_Gesture0-6_980-7.csv und yTest_Gesture0-6_420-7.csv enthalten. Das Verhältnis zwischen Trainings- und Testdaten sollte erfahrungsgemäß 80 zu 20 sein. Wir testen hier etwas strenger und haben mehr Testdaten hinzugefügt.

Für die Erzeugung von Trainings- und Testdaten gibt es ein Programm im Lego-Mindstorms-Projekt. Damit können Sie selbst Trainingsdaten für ihren EV3 erzeugen. Dazu setzen Sie sich vor den Roboter mit ausgestrecktem Arm und führen die Gesten aus. Das Programm „createData“ nimmt jeweils 25 Gesten in die Datei „measures.rtf“ auf. Daraus erstellen sie dann die csv-Dateien auf dem PC, nachdem Sie die rtf-Datei auf den PC geladen haben.

Legen Sie eine leere csv-Datei mit einem Editor an und kopieren Sie die 25 Zeilen mit jeweils 32 Werten aus der measures.rtf-Datei in die csv-Datei. Achten Sie darauf, dass jede Zeile aus 32 Werten besteht, die durch ein Leerzeichen getrennt sind. Jede Zeile muss mit einem Zeilenumbruch (Carriage Return und Linefeed-Zeichen) abgeschlossen sein. Sie können auch aus mehreren measures.rtf-Dateien Zeilen in die csv-Datei einfügen. Das hängt ganz davon ab, wie viele Trainings- oder Testdaten Sie erzeugen wollen.

Training

Das Trainingsprogramm könnten wir in Schritt 3 2 komplett selbst entwickeln. Dies wäre aber viel zu aufwendig. Es gibt unterschiedliche KI-Programmbibliotheken, die kostenlos und sehr leistungsfähig sind. Darunter sind beispielsweise CNTK von Microsoft, Cafe von Facebook und Theano von Universitäten in Montreal. Wir verwenden hier das sehr weit verbreitet genutzte TensorFlow von Google mit einer zusätzlichen API, die Keras heißt.

Letztere vereinfacht die Verwendung von TensorFlow erheblich. Python dient als Programmiersprache, die sich im KI-Bereich immer mehr durchsetzt. Eine Installationsanleitung für alle benötigten Tools und Bibliotheken befindet sich in der Readme-Datei des GitHub-Repository zu diesem Projekt.

Das Trainingsprogramm heißt *train_NN_Keras.py*. Es führt ein komplettes Training durch und erzeugt die Modelldateien für den Roboter im rtf-Format. Das Listing zeigt die Schritte des Trainings. Aus Platzgründen zeigen wir hier nur die wichtigsten Programmzeilen.

Das Listing zeigt die Vorbereitungsschritte 1 bis 3 für das Training: Nachdem die notwendigen Bibliotheken importiert sind, laden wir im ersten Schritt die Trainings- und Testdaten. Danach normalisieren wir alle Merkmalsvektoren der Trainings- und Testdaten in Schritt 2. Normalisierung heißt hier, dass alle Daten als Gleitkommazahlen auf den Zahlenbereich von 0 bis 1 abgebildet werden. Da das Roboterprogramm beim Scannen der Gesten Integer-Zahlen zwischen 1 und 75 liefert, teilen wir alle Werte durch 75. TensorFlow erzielt mit normalisierten Daten bessere Ergebnisse.

Schritt 3 legt unser neuronales Netz an. Die 20 Neuronen der versteckten Schicht wurden experimentell ermittelt: so klein wie möglich und so groß wie nötig. Ist die versteckte Schicht zu klein, erzielen wir keine ausreichende Klassifikationsgüte. Ist diese Schicht zu groß (oder fügen wir noch weitere versteckte Schichten ein), reicht die Rechenleistung des EV3-Roboters im Betrieb nicht mehr aus, um die Geste in weniger als 1 Sekunde zu klassifizieren.

Schritt 3 erzeugt im Weiteren das neuronale Netz in der Variablen „model“. Wie im ersten Teil beschrieben, werden die Schichten sequentiell aneinandergehängt und alle Neuronen einer Vorgängerschicht mit denen der Nachfolgeschicht mit der Methode „Dense“ verbunden. Als Aktivierungsfunktionen legen wir für die versteckte Schicht „ReLU“ (Rectified Linear Unit) fest.

Damit der Lernprozess Zwischenergebnisse auswerten kann, wählen wir die „softmax“-Funktion für die Ausgabeschicht. Sie wandelt den Ergebnisvektor mit 7 Werten so um, dass sich alle Werte zu 1 aufsummieren. Jeder einzelne Wert stellt eine Wahrscheinlichkeit dar, welche Geste vom Neuronalen Netz erkannt wurde.

Schritt 3 endet mit der Konfiguration des Lernprozesses mit „compile“. Die gewählte Loss-Funktion und der Adam-Optimizer eignen sich besonders für die Klassifizierung von Merkmalsvektoren. Eine Loss-Funktion berechnet anhand der Vorhersagen des Netzes und des tatsächlichen Wertes einen Verlustwert. Diese gilt es zu minimieren. Je kleiner der Wert der Loss-Funktion für die Trainingsdaten ist, desto besser wird das Trainingsergebnis. Der Lernprozess richtet sich daher am Wert der Loss-Funktion aus. Die hier gewählte Funktion `categorical_crossentropy` zu dessen Berechnung eignet sich für unser Modell – es gibt noch ein gutes Dutzend anderer Funktionen. Der Verlustwert dient zur Feinabstimmung der einzelnen Gewichtungen bei den Neuronen.

Hier kommt der Optimizer ins Spiel – der eigentlich wichtigste Hauptalgorithmus beim Machine Learning. Während des Lernprozesses passt er die einzelnen Gewichtungen so an, dass er für den Verlustwert ein Minimum findet. Mit jedem neuen Merkmalsvektor verbessert er den Wert. Das Verfahren nennt man auch Backpropagation-Algorithmus, weil die Ergebnisse des Ausgangs auf die Eingangsdaten Einfluss haben – wie in der Regelungstechnik.

Der hier gewählte Adam-Optimizer arbeitet mit einer variablen Lernrate. Die Lernrate legt vereinfacht gesagt die Größe der Anpassungen der Gewichte für jeden neuen Schritt fest. Während dieser Anpassung kann sich der Verlustwert nämlich zwischenzeitlich wieder erhöhen, sodass die Erkennungsrate schlechter wird – so wie bei der Suche lokaler Minima bei Funktionen. Ist die Lernrate zu groß, lernt das Netz zwar schneller, aber es kann lokale Minima übersehen. Mit der Adaptiven Lernrate (adaptive moment estimation) kann sich Adam dem Optimum bes-

ser nähern, da er sein Training an die vorhandenen Trainingsdaten und den momentanen Status des Modells anpasst. Als Metrik verwendet das Programm die Klassifikationsgüte („accuracy").

Alle Trainings- und Testdaten werden in Schritt 4 an die Funktion „fit" übergeben. Die „batch_size" legt fest, wie viele Trainingsdaten auf einmal im Training verarbeitet werden. Je mehr es pro Batch sind, desto schneller geht das Training. Je weniger es sind, desto höher ist die Wahrscheinlichkeit, dass die Klassifikationsgüte des Modells hoch ist. Ähnlich verhält es sich mit der Anzahl der Epochen. Eine Epoche ist eine Iteration über die gesamten Trainingsdaten beim Training. Je öfter man iteriert, desto besser kann die Klassifikationsgüte ausfallen – jedoch ohne Garantie!

Abschließend errechnet Schritt 5 die Klassifikationsgüte und gibt uns damit die Qualität des erzeugten Modells aus. Schritt 6, der aus Platzgründen nicht abgedruckt ist, speichert die Gewichtungen in Dateien, um sie später in das Modell für den Lego-Roboter zu importieren. Die vier Dateien heißen NNweights_h1.rtf, NNbiases_b1.rtf, NNweights_out.rtf und NNbiases_out.rtf.

Trainieren Sie das Modell, indem Sie `python train_NN_Keras.py` aus dem Verzeichnis mit einer Eingabeaufforderung aufrufen, wo das Programm liegt. Das wiederholen Sie viele Male, um zu sehen, wie sich die Klassifikationsgüte bei jedem Mal ändert. Dass sie sich ändert liegt daran, dass die Gewichtungen der Neuronen mit Zufallszahlen

train_NN_Keras.py

```
import keras

from keras.models import Sequential
from keras.layers import Dense
from keras.optimizers import Adam
from numpy import loadtxt, savetxt, reshape
```

Die wichtigsten Python-Bibliotheken und Klassen werden importiert. Numpy ist eine Mathematik-Bibliothek.

```
# Schritt 1: Import Training Data
print('Lese xTrain- und yTrain-Daten')
xTrain= loadtxt('Data/xTrain_Gesture0-6_980-32.csv')
yTrain= loadtxt('Data/yTrain_Gesture0-6_980-7.csv')
xTest= loadtxt('Data/xTest_Gesture0-6_420-32.csv')
yTest= loadtxt('Data/yTest_Gesture0-6_420-7.csv')
```

Die Trainingsdaten werden aus den csv-Dateien im Unterordner Data in (Numpy-) Arrays eingelesen.

```
# Schritt 2: Normalisierung
xTrain /= 75
xTest /= 75
```

Die gemessenen Daten der Infrarotsensoren liegen zwischen 0 und 75 und müssen auf einen für Keras akzeptablen Bereich zwischen 0 und 1 normalisiert werden.

```
# Schritt 3: Definition der neuronalen Netzes
n_input = 32   #Eingangsschicht
n_hidden_1 = 20 #Versteckte Neuronenschicht
n_classes = 7  #Ausgabeschicht
```

Der Aufbau des neuronalen Netzes wird definiert: Es gibt einen Input-Layer mit 32 Neuronen, eine Zwischenschicht mit 20 Neuronen und eine Ausgabeschicht mit 7 Neuronen.

```
model = Sequential() #Ein Layer nach dem anderen
model.add(Dense(n_hidden_1, activation='relu',
                input_shape=(n_input,)))
model.add(Dense(n_classes, activation='softmax'))
model.summary() #Modelleigenschaften ausgeben
model.compile(loss='categorical_crossentropy',
              optimizer=Adam(lr=0.001),
              metrics=['accuracy'])
```

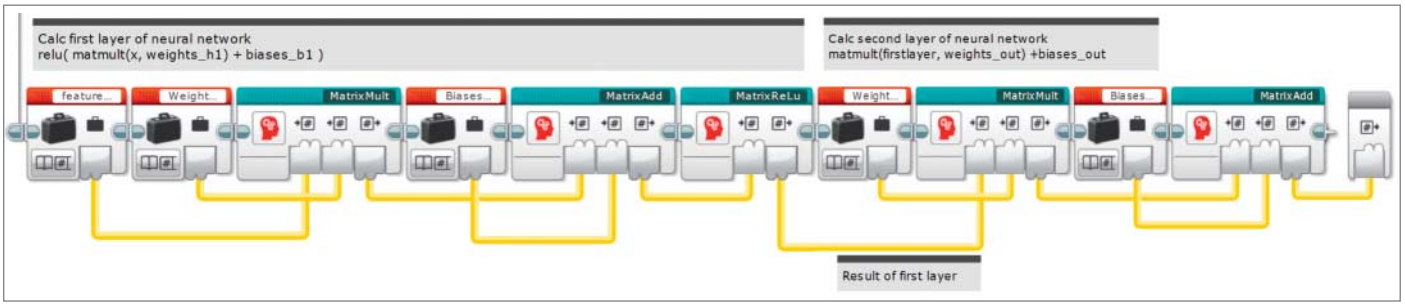
Das Model des neuronalen Netzes wird zusammengebaut und seine Lern-Eigenschaften festgelegt.

```
#Schritt 4: Training
model.fit(xTrain, yTrain, batch_size=1, epochs=200,
          verbose=2, validation_data=(xTest, yTest))
duration = (dt.datetime.now() - start)
print("\nDauer: " + str(duration))
```

Das eigentliche Training und der Test bestehen bei Keras nur aus einem Befehl. Zusätzlich wird noch die Zahl der Durchläufe angegeben.

```
#Schritt 5: Evaluierung
print('Klassifikationgüte: ' +
         repr(model.evaluate(xTest, yTest)[1]))
```

Am Schluss gibt das Skript die Klassifikationsgüte aus. Werte um die 95 Prozent sind sehr gut.



4 **Mit den von uns definierten Blöcken für Matrizenoperationen bauen Sie auf nur einer Bildschirmseite ein Lego-Programm, das das neuronale Netz berechnet.**

vorbelegt werden, sodass sie bei einem Start vielleicht gerade gut passen, um ein gutes Ergebnis zu erzielen, und beim nächsten Mal gerade nicht. Eine Klassifikationsgüte von über 90 % ist ausreichend. Das Modell im GitHub-Repository hat eine Klassifikationsgüte von 94,8 %.

Das neuronale Netz im Roboter

In der Übersicht 2 sind wir nun bei Schritt 7 und 8 angekommen. Der Code zur Berechnung des neuronalen Netzes ist bereits im Lego-Mindstorms-Projekt im GitHub-Repository vorhanden (Gestur3Bot_V1.ev3). Laden Sie das Projekt in den Roboter. Nun fehlen nur die Gewichtungen, die Sie entweder selbst mit dem Trainingsprogramm erstellt haben oder aus dem GitHub-Repository kopieren.

Auf den Legostein kopieren Sie die Dateien, indem Sie den Browser der Lego-Entwicklungsumgebung verwenden. Klicken Sie dazu den Projektordner an und laden alle 4 Dateien (NNweights_h1.rtf, NNbiases_b1.rtf, NNweights_out.rtf und NNbiases_out.rtf.) auf den Roboter. Dann können Sie das Programm „main“ auf dem Roboter starten. Führen Sie die Gesten in der Geschwindigkeit aus, wie auf dem Video demonstriert.

Die Implementierung des neuronalen Netzes im Roboter setzt die Formeln aus dem ersten Teil des Artikels um. Sie ist erstaunlich kompakt, da sie bei der Berechnung der beiden Schichten auf die Operationen Matrizenmultiplikation und Matrizenaddition zurückgreifen kann 4.

Das Ergebnis des neuronalen Netzes finden wir in der Ausgabeschicht. Die erkannte Geste (Klasse) ist der Index, dessen zugehöriger Wert am größten ist. Ist also der Wert mit Index 2 in der Ausgabeschicht am größten, so hat das neuronale Netz Geste 2 erkannt.

```
>>> y2 = model.predict_proba(xnew,verbose=1)
1/1 [==============================] - 0s 396us/step
>>> print (y2)
[[1. 0. 0. 0. 0. 0. 0.]]
```

Der Ergebnisvektor enthält die Wahrscheinlichkeiten, welche Geste gerade erkannt wurde. Hier Geste 0.

```
Using TensorFlow backend.
Training eines Neuronalen Netzes mit KERAS(V1.1)

Lese xTrain- und yTrain-Daten

_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
dense_1 (Dense)              (None, 20)                660
_________________________________________________________________
dense_2 (Dense)              (None, 7)                 147
=================================================================
Total params: 807
Trainable params: 807
Non-trainable params: 0
_________________________________________________________________
Train on 980 samples, validate on 420 samples
Epoch 1/200
 - 1s - loss: 1.2831 - acc: 0.5276 - val_loss: 0.8382 - val_acc: 0.7476
Epoch 2/200
 - 1s - loss: 0.6541 - acc: 0.7714 - val_loss: 0.5612 - val_acc: 0.8429
Epoch 3/200
 - 1s - loss: 0.4792 - acc: 0.8214 - val_loss: 0.4682 - val_acc: 0.8262
```

Der Trainingslauf des Python-Skripts mit 200 Epochen

Klassifikationsgüte verbessern

Zum Abschluss wollen wir noch einen Blick auf Möglichkeiten werfen, die Klassifikationsgüte zu verbessern. Da sie während eines Trainings nicht stetig steigt, ist es interessant, zunächst ihren Verlauf während eines Trainings zu visualisieren. Tensorboard von TensorFlow macht genau das. Im GitHub-Repository finden Sie das Programm train_NN_Keras_Tensorboard.py, das den Verlauf der Klassifikationsgüte (val_acc) während des Trainings als Kurve darstellt. Beachten Sie bitte die Hinweise im Programmkopf.

Das Programm train_NN_Keras_opt.py prüft nach jeder Epoche die Klassifikationsgüte und merkt sich das Modell, falls es sich verbessert hat. So erreichen wir, dass das beste Modell aus einem Trainingslauf als Ergebnis vorliegt. Denn das Modell der letzten Epoche ist oft nicht das beste eines Trainings.

Spielen Sie bei allen drei Programmen ein wenig mit den Parametern `batch_size` und Anzahl der Epochen (`epochs`) und beobachten Sie, wie sich das Training und das Ergebnis verändert. Verändern Sie auch die Anzahl der Neuronen in der versteckten Schicht. Teilen Sie uns mit, falls Sie eine Klassifikationsgüte von über 95 % erreichen. Viel Spaß beim Forschen und make it so! —*dab*