DIE ZEITSCHRIFT FÜR DIE MICROSOFT DEVELOPER COMMUNITY

VisualStudio1.de

65
2017

DE 8,50 EUR
A/CH 9,50 EUR
EU 9,50 EUR

BEWEGTE BILDER IM GIF-FORMAT

BLUESCREEN MIT QR-CODE

SCHAU DER WELLENFORM AUFS MAUL!

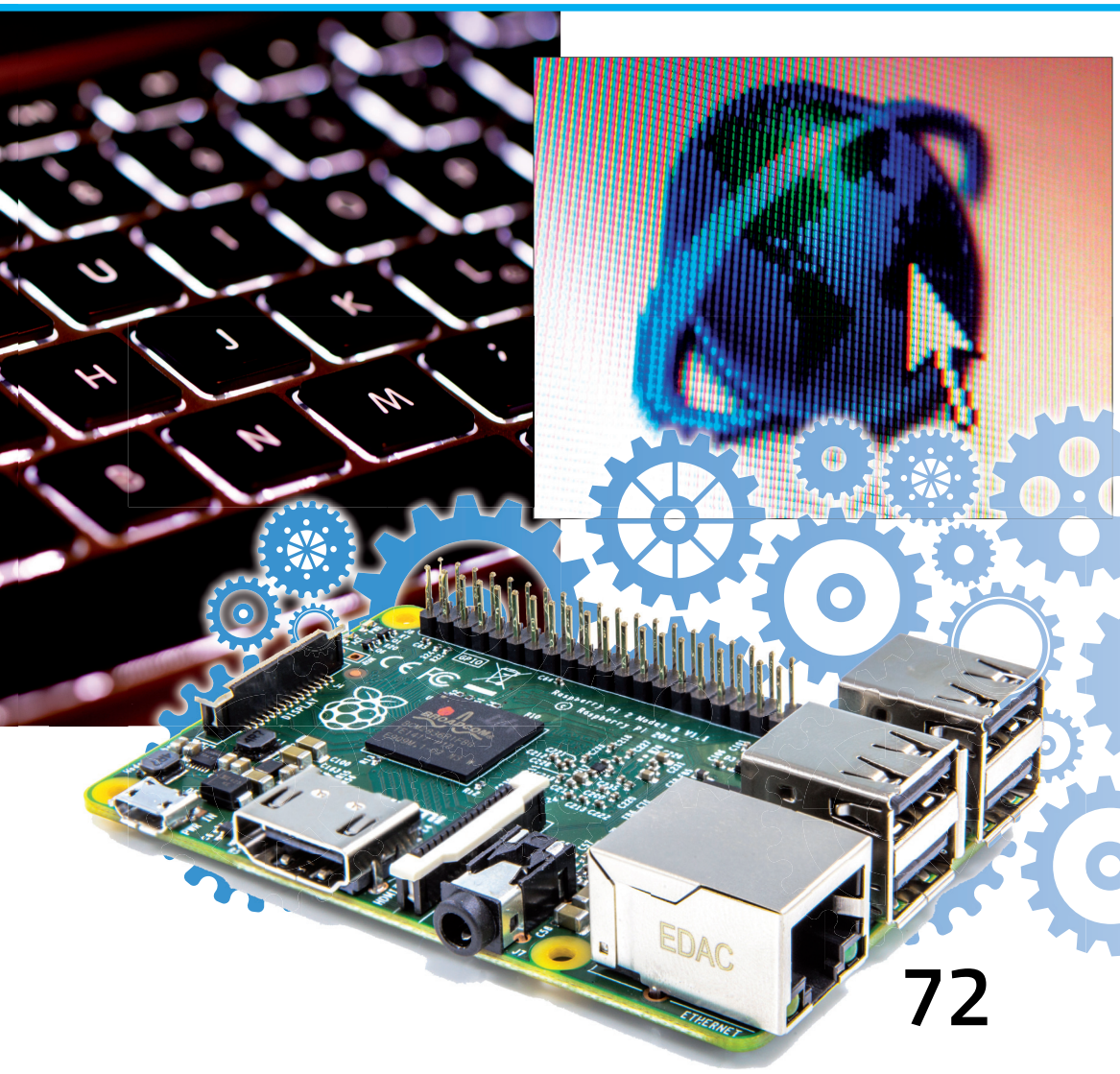
TESTABDECKUNG MIT NCOVER

FLUENT ASSERTIONS

MASCHINELLES LERNEN

(01)41960661085006

INHALT AUSGABE 65/2017



„DREH DIE KAPSEL BITTE WEITER, HAL!"

Neuronale Netze, maschinelles Lernen, Deep-Learning & Co. sind in aller Munde. Doch was genau ist eigentlich ein neuronales Netz und wie funktioniert es? Golo Roden zeigt, dass gängige Schulmathematik ausreicht, um das Konzept zu verstehen.

Autor: GOLO RODEN

Der klassische Weg, Computern neue Fähigkeiten beizubringen, besteht darin, sie zu programmieren. Eine Alternative könnte darin bestehen, Computern das Lernen beizubringen, so dass sie eigenständig Probleme lösen können. Doch stellt sich die Frage, wie das funktioniert: Wie lässt sich Lernen lernen?

Bemerkenswert an der Frage ist, dass jedes Lebewesen vom ersten Augenblick an lernt und diese Fähigkeit „einfach so" mitbringt. Computer übertreffen zwar sämtliche Lebewesen im Hinblick auf ihre Rechengeschwindigkeit, aber lernen können sie nicht von Haus aus. Neuronale Netze sind ein Ansatz, Computern das Lernen an sich beizubringen.

Allerdings sollte man keine Wunder erwarten: Bis zur Leistung einer künstlichen Intelligenz wie dem Computer HAL 9000 aus dem Film „2001: Odyssee im Weltraum" ist es noch ein weiter Weg. Das gleiche gilt für die regelmäßig beschriebene Dystopie, in der eine künstliche Intelligenz mit eigenem Bewusstsein der Menschheit den Garaus macht.

Ein neuronales Netz ist keine allmächtige Wundertechnologie, sondern lediglich eine andere Art, einen Algorithmus zu formulieren, der ein spezifisches Problem löst. Neuronale Netze sind in ihrer Grundform zwar universell einsetzbar, sie müssen aber auf konkrete Probleme geprägt beziehungsweise trainiert werden.

NEURONEN UND DIE SIGMOID-FUNKTION

Die Grundidee für neuronale Netze ist ähnliche wie bei genetischen Algorithmen an die Biologie angelehnt. In der Bologie ist ein Neuron eine Nervenzelle, die über Dendriten und ein Axon die Signale anderer Neuronen empfangen und eigene Signale an diese weiterleiten kann.

Überträgt man das Konzept auf die Informatik, lässt sich ein Neuron als Funktion beschreiben, die mehrere Parameter und einen Rückgabewert hat. In Abhängigkeit der Eingangssignale berechnet sie ein Ausgangssignal. Damit man Neuronen miteinander kommunizieren lassen kann, muss die Kommunikation zwischen ihnen standardisiert sein.

Dazu wird der Wertebereich auf Werte zwischen 0 und 1 begrenzt. Man benötigt also zunächst eine Funk-

tion, die einen beliebigen numerischen Wert auf diesen Bereich reduziert. Dabei ist die Mitte, das heißt der Bereich um die 0.5, interessanter als die Ränder: Das liegt daran, dass eine lediglich geringe Abweichung von einem gewünschten Wert große Auswirkungen haben kann, es aber in der Regel gleichgültig ist, um wie viel eine große Abweichung danebenliegt.

Das lässt sich an einem konkreten Beispiel gut veranschaulichen: Sind auf einer Straße als Höchstgeschwindigkeit 130 km/h zugelassen, spielt es durchaus eine Rolle, ob man mit 129, 130 oder 131 km/h unterwegs war. Ob man sich hingegen mit 220 oder 250 km/h fortbewegt hat, ist verhältnismäßig egal: Beides ist deutlich zu hoch.

Genau das leistet die sogenannte [Sigmoid-Funktion](https://de.wikipedia.org/wiki/Sigmoidfunktion), die sich beispielsweise in JavaScript wie folgt berechnen lässt:

```
const sigmoid = function (t) {
  return 1 / (1 + Math.pow(Math.E, -t));
};
```

Um die Geschwindigkeit zu normieren, muss man nun zunächst 130 abziehen. Dann ergibt eine Eingabe von 130 km/h einen Wert von `0.5`, da gilt:

```
const speed = 130;

const getNormalized = function (speed) {
  return speed - 130;
};

console.log(sigmoid(getNormalized(speed)));
// => 0.5
```

Eine Abweichung von lediglich einem Stundenkilometer nach oben ergibt `0.73`, eine Abweichung nach unten ergibt `0.26`. Wie man sieht, sind die Schwankungen nah um den Nullpunkt sehr groß. Eine Abweichung um 70 (bei 200 km/h) oder 120 (bei 250 km/h) nach oben ergibt aber jeweils `1`. Die Abweichung ist in dem Fall dermaßen groß, dass es keine Rolle mehr spielt, wie groß sie ist.

Geht man nun von der (hypothetischen) Annahme aus, dass zu schnelles Fahren bei Regen noch weitaus schlimmer ist als bei Sonnenschein, könnte man auf die Idee kommen, eine Funktion zu schreiben, die neben der Geschwindigkeit auch das Wetter als Eingabe erwartet und einen gemeinsamen Wert berechnet. Dazu gilt es aber zunächst, das Wetter numerisch zu codieren.

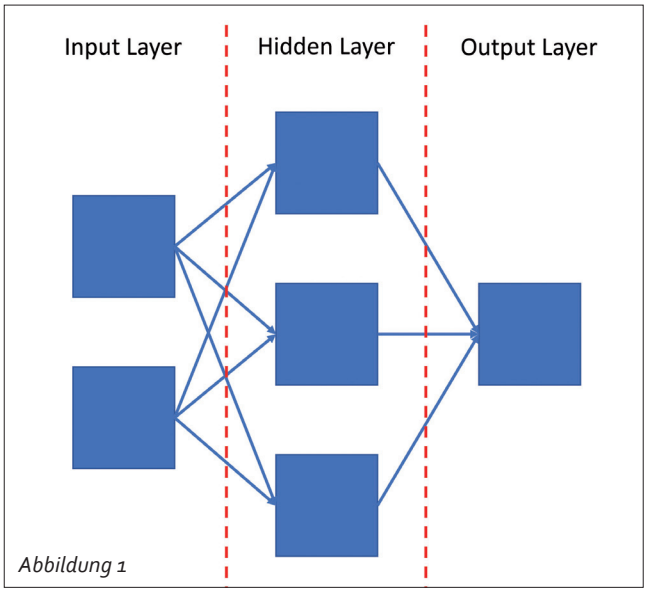


Abbildung 1

Der Einfachheit halber könnte man beispielsweise festlegen, dass Regen dem Wert `1` entspricht, und fehlender Regen dem Wert `-1`. Das lässt sich in dem Fall so einfach codieren, da in dem Beispiel eine rein duale Logik zu Grunde liegt: Entweder es regnet oder es regnet nicht. In der Praxis ist das Codieren von Werten deutlich aufwändiger und erfordert mehr Nachdenken.

Aus den Überlegungen ergibt sich nun eine Funktion, die die beiden Eingaben entgegennimmt, aufsummiert und sie anschließend mit Hilfe der Sigmoid-Funktion in eine normierte Ausgabe überführt:

```
const score = function (speed, isRaining) {
  const normalizedSpeed = getNormalized(speed),
        normalizedWeather = isRaining ? 1 : -1;

  const sum = normalizedSpeed + normalizedWeather;

  return sigmoid(sum);
};
```

Ruft man die Funktion nun mit verschiedenen Werten auf, lässt sich beobachten, wie sich die Ausgabe verändert:

```
score(130, true);  // => 0.73
score(130, false); // => 0.26
score(131, true);  // => 0.88
score(131, false); // => 0.5
```

DIE EINGABEN GEWICHTEN

Wie man sieht, steigt der Wert bei steigender Geschwindigkeit. Außerdem führt Regen zu einem höheren Wert. So weit, so gut. Allerdings weist die Berechnung einen gravierenden Makel auf: Die Geschwindigkeit und das Wetter fließen zu gleichen Teilen in das Ergebnis ein.

Besser wäre es, die Anteile gezielt gewichten zu können. Das ist kein Problem, denn dazu muss lediglich der jeweilige Parameter noch mit einem frei wählbaren Faktor multipliziert werden. Soll beispielsweise die Geschwindigkeit eine doppelt so große Rolle wie das Wetter spielen, könnte man als Faktoren zum Beispiel die Zahlen `1` und `2` wählen:

```
const score = function (speed, isRaining) {
  const normalizedSpeed = getNormalized(speed),
        normalizedWeather = isRaining ? 1 : -1;

  const weightedSum =
    2 * normalizedSpeed +
    1 * normalizedWeather;

  return sigmoid(weightedSum);
};
```

Allgemein formuliert ergibt sich auf dem Weg eine Funktion, die wie folgt aussieht. Die Parameter `weights` und `inputs` sind dabei jeweils Arrays und müssen die gleiche Länge aufweisen. Außerdem müssen die Eingaben zuvor bereits normiert worden sein:

```
const f = function (weights, inputs) {
  let weightedSum = 0;

  for (let i = 0; i < weights.length; i++) {
    weightedSum += weights[i] * inputs[i];
  }

  const output = sigmoid(weightedSum);

  return output;
};
```

VON NEURONEN ZU NEURONALEN NETZEN

Eine solche Funktion ist ein Neuron. Schaltet man nun mehrere dieser Funktionen zu einem Netz zusammen, ergibt sich ein sogenanntes neuronales Netz. Neuronale Netze werden aus Schichten aufgebaut, wobei jede Schicht wiederum aus mehreren Neuronen bestehen kann.

Die Idee dabei ist, dass jedes Neuron einer Schicht als Eingaben die Ausgaben sämtlicher Neuronen der Vorgängerschicht erhält. Seine eigene Ausgabe wird wiederum an alle Neuronen der Nachfolgeschicht weitergereicht.

Klassische neuronale Netze bestehen aus drei Schichten: Die erste Schicht ist der sogenannte I*nput Layer*, die dritte der sogenannte *Output Layer*. Die Schicht dazwischen wird als *Hidden Layer* bezeichnet (siehe Abbildung 1). Verfügt ein neuronales Netz über mehr als einen Hidden Layer, spricht man von *Deep Learning*.

Das Kunststück besteht nun darin, die Gewichte der einzelnen Neuronen in den verschiedenen Schichten so zu wählen, dass sich das gewünschte Ergebnis einstellt. Wie man sich leicht vorstellen kann, ist das bereits bei einfachen Netzen eine mühselige Arbeit, die von Hand kaum zu bewerkstelligen ist.

Daher benötigt man einen anderen Ansatz, um das gewünschte Ergebnis zu erzielen. Den Vorgang, ein neuronales Netz zu konfigurieren, bezeichnet man als Training. Um ein neuronales Netz trainieren zu können, müssen Trainingsdaten zur Verfügung stehen, die das gewünschte Ergebnis beschreiben.

Da alle Geschwindigkeiten kleiner als 130 km/h unproblematisch sind, lässt sich für sie ein einheitlich niedriger Wert festlegen. Bei höheren Geschwindigkeiten wird die Ausgabe dann allerdings interessant:

Speed	Raining	Result
50	false	0.10
50	true	0.10
120	false	0.10
120	true	0.10
130	false	0.10
130	true	0.10
135	false	0.60
135	true	0.70
140	false	0.66
140	true	0.80
200	false	0.90
200	true	0.99

Prinzipiell ließe sich einwenden, dass dafür kein neuronales Netz erforderlich ist, schließlich ist die Kombination der möglichen Eingabevarianten endlich. Der Einwand ist durchaus berechtigt, jedoch übersteigt die Anzahl der möglichen Kombinationen, gerade wenn noch weitere Parameter ins Spiel kommen, rasch die Möglichkeit, die zur Berechnung des Ergebnisses erforderliche Funktion von Hand zu entwickeln.

EIN NEURONALES NETZ TRAINIEREN

Das Training eines neuronalen Netzes läuft nun folgendermaßen ab: Zunächst initialisiert man alle Gewichte mit beliebigen Werten. Dabei spielt es keine Rolle, ob man sie willkürlich wählt oder sie von einem Zufallsgenerator bestimmen lässt. Anschließend übergibt man den ersten Datensatz der Trainingsdaten an das neuronale Netz und lässt es das Ergebnis berechnen.

Das Ergebnis wird höchst wahrscheinlich nicht dem gewünschten Ergebnis entsprechen. Also gilt es nun die Gewichte zu justieren, um dem gewünschten Ergebnis näher zu kommen. Das neuronale Netz „lernt" auf dem Weg mehr über das von ihm erwartete Verhalten. Es wird gewissermaßen konditioniert.

Anschließend übergibt man dem neuronalen Netz den nächsten Datensatz aus den Trainingsdaten und berechnet erneut das Ergebnis, vergleicht es wieder mit dem gewünschten Ergebnis und passt die Gewichte erneut so an, dass das tatsächliche Ergebnis dem erwarteten näher kommt - ohne sich dabei aber wiederum zu weit vom ersten Datensatz zu entfernen.

Der Vorgang wird anschließend vielfach wiederholt. Mit jeder Iteration passt sich das neuronale Netz besser an die Trainingsdaten an. Das tatsächlich berechnete Ergebnis verbessert sich dabei zusehends, und nach einer Weile ist das Netz dann auch in der Lage, realistische Ergebnisse für noch nicht bekannte Werte zu berechnen.

FEHLER VERMEIDEN

Gelegentlich kommt es zu einem Effekt, der als *Overfitting* bezeichnet wird. In dem Fall lernt das neuronale Netz die Trainingsdaten zu gut kennen und passt sich exakt auf sie an. Dabei verliert es die Fähigkeit, aus den Trainingsdaten zu abstrahieren, um auch unbekannte Eingaben sinnvoll verarbeiten zu können.

Daher ist es empfehlenswert, die initial gegebene Menge von Trainingsdaten in zwei Hälften zu teilen, und nur die eine für das Training zu verwenden. Mit der anderen lässt sich die Qualität des Ergebnisses unabhängig kontrollieren.

Wichtig ist auch die Frage, wie man den Fehler überhaupt bewertet. Angenommen, die Trainingsdaten geben als gewünschtes Ergebnis eine 0.8 vor, das neuronale Netz berechnet jedoch lediglich eine 0.6. Wie groß ist dann der Fehler? Auf den ersten Blick könnte man sagen, er sei 0.2, da man zum berechneten Wert von 0.6 noch 0.2 hinzuzählen muss, um zu 0.8 zu gelangen.

Analog wäre der Fehler, der bei einer 0.8 eine 1.0 berechnet, -0.2. Warum das problematisch ist, zeigt sich, wenn man den mittleren Fehler ermitteln will: Das wäre in dem Fall nämlich 0, weil sich 0.2 und -0.2 gegenseitig aufheben! Doch selbstverständlich ist ein Fehler von 0.2 genauso schlecht wie ein Fehler von -0.2. Daher wird der Fehler üblicherweise quadriert, was diesen Effekt verhindert:

```
 0.2 *  0.2 = 0.04
-0.2 * -0.2 = 0.04
```

Außerdem werden durch das Quadrieren größere Fehler stärker gewichtet als kleinere, was das Lernen beschleunigt: Man weiß, dass man in diesem Fall einen größeren Schritt in die richtige Richtung machen kann als bei einem nur sehr kleinen Fehler.

Einen Schritt in die richtige Richtung heißt in dem Zusammenhang, den Fehler zu minimieren. Das funktioniert über das Konzept der mathematischen Ableitung, womit sich die Steigung einer Funktion berechnen lässt.

Nun zeigt sich auch, warum die Sigmoid-Funktion so eine häufig und gern verwendete Funktion im Zusammenhang mit neuronalen Netzen ist: Sie ist stetig und differenzierbar, das heißt, ihre Ableitung lässt sich problemlos berechnen. Prinzipiell lassen sich aber auch andere Funktionen verwenden, solange sie ähnliche Eigenschaften aufweisen.

Nachdem man den Fehler des Ergebnisses berechnet hat, gilt es, die Gewichte für den Output Layer zu berechnen und den Fehler auf dessen Gewichte anteilsmäßig zu verteilen. Anschließend wendet man auch das Vorgehen wieder iterativ an und passt die Gewichte des Hidden Layers an, und schließlich jene des Input Layers. Das Vorgehen wird als Back Propagation bezeichnet. Je mehr Layer ein Netz enthält und je mehr Neuronen beteiligt sind, desto aufwändiger wird diese Berechnung.

DIE BERECHNUNG BESCHLEUNIGEN

Eine spannende Frage ist, wie sich das beschleunigen lässt. Dazu gibt es zwei Überlegungen: Zum einen lassen sich die Berechnungen an sich beschleunigen. Da sich die Berechnungen als Operationen der Vektor- und Matrizenrechnung abbilden lassen, kann hierbei auf entsprechend optimierte Bibliotheken zurückgegriffen werden.

Auf dem Weg lassen sich viele unnötige Schleifen einsparen. Das Ersetzen von schleifenbasiertem und prozeduralem Code durch optimierte Berechnungen mit Vektoren und Matrizen wird als Vektorisierung bezeichnet.

Die zweite Überlegung ist, wie viele Neuronen und Layer überhaupt verwendet werden sollten. Die Anzahl der Neuronen im Input Layer und Output Layer lässt sich leicht festlegen: Die Anzahl der Variablen entspricht der Anzahl der Neuronen im Input Layer, die Anzahl der gewünschten Ergebnisvariablen entspricht der Anzahl der Neuronen im Output Layer.

Im vorliegenden Beispiel, wo die Geschwindigkeit und das Wetter auf einen einzigen numerischen Wert abgebildet werden sollen, bedeutet das, dass das neuronale Netz über zwei Neuronen im Input Layer und ein Neuron im Output Layer verfügt.
Doch was ist mit dem Hidden Layer beziehungsweise den Hidden Layers? Wie viele Schichten sollte man verwenden, und wie viele Neuronen sollte jede einzelne Schicht aufweisen? So unbefriedigend das klingen mag, ist der beste Ansatz, verschiedene Varianten auszuprobieren und die einzelnen Ergebnisse zu vergleichen.

Dann lässt sich auf Grund der Qualität der Ergebnisse und der benötigten Rechendauer bestimmen, welchen Pfad man weiterverfolgen sollte und wo sich weiteres Training lohnt.

FAZIT

Wie leicht zu sehen ist, sind neuronale Netze keine schwarze Magie. Sie sind lediglich eine Reihe von für sich genommen sehr einfachen Funktionen, die parallel und in Reihe geschaltet werden. Insbesondere haben neuronale Netze kein Bewusstsein und haben auch keine eigenen Wünsche, Bedürfnisse oder Ideen. Daher ist die Angst vor einer Machtübernahme durch künstliche Intelligenz unbegründet, zumindest im Hinblick auf neuronale Netze.

Trotzdem können diese Systeme beeindruckendes leisten. Das Hauptproblem besteht meistens allerdings darin, die Eingangsdaten überhaupt erst einmal in ein passendes Format zu transformieren. Für das gezeigte Beispiel war das einfach, doch wie geht man mit Bildern oder Audiodaten um? Das sind im praktischen Gebrauch die eigentlich spannenden Fragen, bei denen es sich häufig lohnen kann, einen Datenexperten zu Rate zu ziehen, der Erfahrung mit maschinellem Lernen hat.

Wer kein neuronales Netz von Grund auf selbst entwickeln will, kann auf eins der zahllosen Systeme am Markt zurückgreifen. Dabei muss es je nach Anwendungsfall nicht immer um ein hochkomplexes System wie [TensorFlow](https://www.tensorflow.org/) handeln, es finden sich durchaus auch kleine leistungsfähige Module in der Programmiersprache der Wahl.