WIKIPEDIA

Long short-term memory

Long short-term memory (**LSTM**, deutsch: **langes Kurzzeitgedächtnis**) ist eine Technik, die zur Verbesserung der Entwicklung von künstlicher Intelligenz wesentlich beigetragen hat.

Beim Trainieren von künstlichen neuronalen Netzen werden Verfahren des Fehlersignalabstiegs genutzt, die man sich wie die Suche eines Bergsteigers nach dem tiefsten Tal vorstellen kann. Bei mehreren vertiefenden Schichten kann dies zu kurz greifen, so wie ein vergesslicher Bergsteiger beim Abstieg im ersten besten Tal landet und sein Dorf in einem tieferen Tal nicht finden kann. Das LSTM-Verfahren löst dieses Problem, indem es für eine LSTM-Zelle zur besseren Erinnerung drei Torsorten verwendet: Ein Eingangstor (Input Gate), ein Merk- und Vergesstor (Forget Gate) und ein Ausgangstor (Output Gate). LSTM ermöglicht auf diese Weise im Gegensatz zu herkömmlichen rekurrenten neuronalen Netzen eine Art Erinnerung an frühere Erfahrungen: Ein Kurzzeitgedächtnis, das lange anhält.

1997 wurden LSTM-Netze von Sepp Hochreiter und Jürgen Schmidhuber in einer Veröffentlichung vorgestellt[1] und 2000 von Felix Gers und seinem Team verbessert. Seit etwa 2016 feiert LSTM bedeutende Erfolge, da seitdem große Datenmengen zum Training genutzt werden können, weitere Verbesserungen der LSTM-Technik durchgeführt wurden, hinreichend leistungsfähige Rechner zur Verfügung stehen und Grafikprozessor-Programmierung angewendet wird.

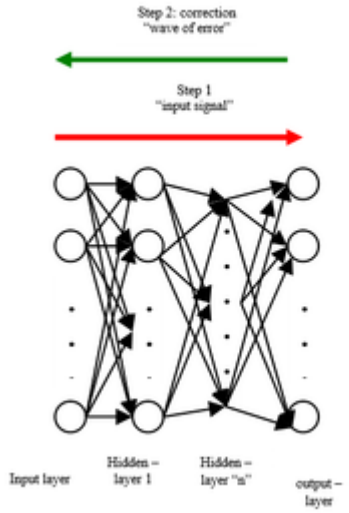
Neuronale Netze mit vielen Schichten sind extrem lernfähig. LSTM sorgt dafür, dass genau solche mehrschichtigen Netze gut funktionieren können. Dies hat einen Durchbruch bei der künstlichen Intelligenz ermöglicht.

Inhaltsverzeichnis

Verschwindender oder explodierender Gradient

Eine Möglichkeit, künstliche neuronale Netze zu trainieren, ist die Fehlerrückführung. In der frühen Trainingsphase macht ein Netz beispielsweise bei der Mustererkennung manches falsch: Auf einem Bild mit Katze soll eine Katze erkannt werden und kein Hund. Zur Korrektur des Fehlers werden die Auslöser der Abweichungen (Fehler) zwischen erzeugter Zuordnung (Hund) und Lösungszuordnung (Katze) zurückverfolgt und wiederholt steuernde Faktoren (Gewichte) in den Schichten des Netzes jeweils so verändert, dass die Zuordnungsfehler kleiner und kleiner werden. Im sogenannten Gradientenverfahren wird dieser Fehler minimiert: Die Zahlen in den steuernden Gewichten werden

neu justiert. Neuronale Netze bestehen aus hintereinandergeschalteten Modulen, die klassischerweise jeweils nur eine einzige Aktivierungsfunktion besitzen, die dafür sorgt, dass die Ausgabe zwischen 0 und 1 liegt. Bei jeder Fehlerkorrektur wird das Fehlersignal durch die Ableitung der Aktivierungsfunktion bestimmt. Durch diese Ableitung wird die Abstiegssteigung und die Richtung bestimmt, mit der das Fehlertal ermittelt wird. Sepp Hochreiter erkannte 1991, dass dieses bis dahin übliche Verfahren bei mehrschichtigen Netzen ungeeignet ist.[2] Je weiter nämlich der Fehler im Prozess (von hinten nach vorne gesehen) berechnet wird, desto öfter wird der Skalierungsfaktor mit dem Fehlerterm multipliziert. Wenn der Faktor (hier der Spektralradius einer Gewichtsmatrix) stets kleiner als 1 ist, verschwindet der Fehler und führt zu ineffektiven Gewichtsaktualisierungen: Denn wenn Zahlen zwischen 0 und 1 miteinander multipliziert werden, so ist das Produkt kleiner als der kleinere der beiden Faktoren. Ein ursprünglich hoher Wert verschwindet also auf lange Sicht. Wenn die Faktoren andererseits größer als 1 wären, würde der Fehlerwert auf die Dauer explodieren.



Im ersten Schritt wird vorwärts ein Signal erzeugt (roter Pfeil). Dann wird (grün) als Fehlerjustierung rückwärts die Gewichtung korrigiert.

Die Module in der Mitte des Netzes, sogenannte Hidden Layer, die der Eingabeschicht näher sind als der Ausgabeschicht, werden also bei der (rückwärts berechneten) Fehlerjustierung zu wenig berücksichtigt. Das führt dazu, dass sie kaum trainiert werden, so als wenn beim Fußball nur die Stürmer dazulernen, wenn es um das Toreschießen geht, nicht jedoch die Mittelfeldspieler oder Verteidiger.

Drei Gates und eine innere Zelle

Um dieses Problem zu lösen, wurde ein LSTM-Modul entworfen, das einen relativ konstanten und anwendbaren Fehlerfluss ermöglicht.[1] Man schaut sich genau an, welche Informationen in die innere Zelle hineinlaufen und hinauslaufen sollen. Das LSTM hat die Fähigkeit, Informationen zum Zellzustand zu entfernen oder hinzuzufügen, sorgfältig reguliert durch Strukturen, die Tore oder Gates genannt werden. LSTM-Module sind zwar ebenso wie herkömmliche Module kettenartig hintereinandergeschaltet, aber sie haben intern eine andere Struktur: Die zusätzlichen Gates sind eine Möglichkeit, Informationen optional durchzulassen.

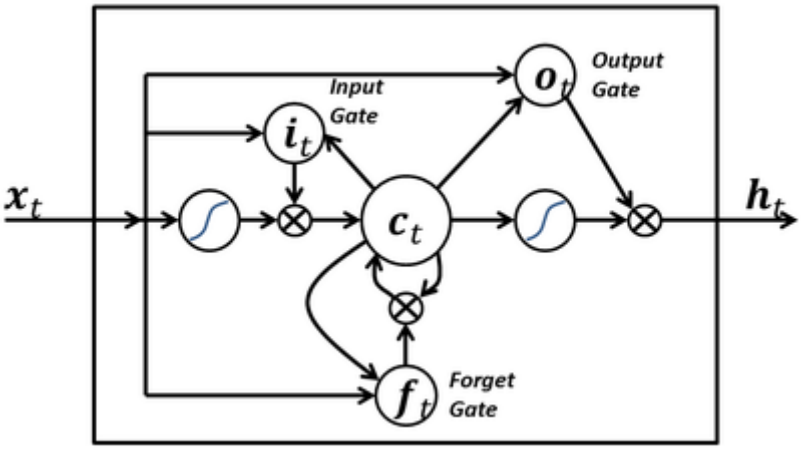
Statt einer einzigen neuronalen Funktion im LSTM-Modul gibt es vier, die auf eine ganz besondere Art und Weise miteinander interagieren. Ein LSTM-Modul enthält die genannten drei Gates und eine innere Zelle. Kurz gesagt steuert

- das Input Gate das Ausmaß, in dem ein neuer Wert in die Zelle fließt,
- das Forget Gate das Ausmaß, in dem ein Wert in der Zelle verbleibt bzw. vergessen wird[3][4] und
- das Output Gate das Ausmaß, in dem der Wert in der Zelle zur Berechnung für das nächste Modul der Kette verwendet wird.

Diese Netzelemente werden mit sigmoiden neuronalen Funktionen und diversen Vektor- und Matrixoperationen verbunden und ineinander überführt.

Aufbau eines LSTM

Es gibt verschiedene Arten von LSTM-Architekturen. Üblich ist besonders bei der Bildverarbeitung das convolutionale LSTM-Netz, das hier skizziert wird.[5] Es unterscheidet sich vom bloßen Peephole LSTM, das die Matrixmultiplikation verwendet, dadurch, dass die Aktivität jedes Neurons über eine diskrete Faltung (daher der Zusatz *convolutional*) berechnet wird. Intuitiv wird dabei schrittweise eine vergleichsweise kleine Faltungsmatrix (Filterkernel) über das Inputbild bewegt. Guckloch (Peephole) heißen diese Netze, weil die Gates den Zellstatus *sehen* können, also auch die Informationen aus der Zelle verarbeiten. Index *t* ist jeweils der aktuelle Durchlauf, *t-1* bezeichnet den vorherigen Durchlauf. *d* und *e* sind jeweils die Anzahlen der Spalten und Zeilen von Vektoren und Matrizen.



Grober Aufbau eines LSTM-Moduls mit der inneren Zelle im Zentrum. Die ⊗-Symbole repräsentieren hier den Faltungsoperator. Die großen Kreise mit S-artiger Kurve sind die Sigmoidfunktionen. Die Pfeile, die von der Zelle jeweils zu den Gates zeigen, sind die Gucklochinformationen vom letzten Durchlauf.

Der Datenfluss zwischen den verschiedenen Gates und ihrer inneren Zelle ist durch Vektor- und Matrizenoperationen bestimmt. Zunächst wird hier die mathematische Struktur des Forget Gates beschrieben. f_t ist der dazugehörende *e*-stellige Aktivierungsvektor:

$$f_t = \sigma_g(W_f * x_t + U_f * h_{t-1} + V_f \circ c_{t-1} + b_f)\,.$$

x_t ist der *d*-stellige Inputvektor. In der Kette aufeinander folgender Neuronen ist er (zusammen mit dem Outputvektor h_{t-1} des vorigen Durchlaufs) die Schnittstelle zum vorher in der Kette agierenden Neuron. Die drei $e \times d$-stelligen Gewichtsmatrizen (weight matrices) W, U, V bilden den wertvollen Teil jedes Netzes, weil sie das Trainingswissen enthalten. b ist der Bias-Vektor. Wenn kein starker Input von anderen Einheiten erfolgt, dann stellt das Bias sicher, dass die Einheit bei starkem Gewicht aktiv bleibt und bei schwachem inaktiv. σ_g stellt eine Sigmoidfunktion der Gates dar, die nichtlinear Werte zwischen 0 und 1 aus dem Ganzen bildet.

Es gibt hier drei verschiedene Arten von Matrizenoperatoren:

- +: Matrizenaddition
- ∘: Hadamard-Produkt (für die Gucklochinformationen des vorigen Durchlaufs)
- ∗: Matrizenmultiplikation.

Diese formelhaften Darstellungen erscheinen zwar kompliziert, aber das tatsächliche Rechnen übernehmen die jeweiligen Programmbibliotheken der Anbieter für KI.

Hier die Struktur des Aktivierungsvektors i_t vom Input Gate und o_t, dem Vektor des Output Gates, sie entsprechen beide dem Aufbau des Forget Gate Vektors:

$$i_t = \sigma_g(W_i * x_t + U_i * h_{t-1} + V_i \circ c_{t-1} + b_i)$$
$$o_t = \sigma_g(W_o * x_t + U_o * h_{t-1} + V_o \circ c_{t-1} + b_o)$$

Der Zellzustand ist so etwas wie ein Förderband. Die Information verläuft geradlinig über die gesamte Kette, mit nur geringen linearen Wechselwirkungen. Die innere Zelle mit dem Zellstatusvektor c_t hat folgenden Aufbau:

$$c_t = f_t \circ c_{t-1} + i_t \circ \sigma_c(W_c * x_t + U_c * h_{t-1} + b_c)\,.$$

Für die Sigmoidfunktionen σ_c und σ_h wird üblicherweise der hyperbolische Tangens (auch: tanh) verwendet. h_{t-1} ist der Outputvektor (des vorigen Durchlaufs, nicht im groben Schaubild zu sehen).

Die Anfangswerte für $c_0$und $h_0$werden jeweils mit Nullvektoren initialisiert. Der Outputvektor berechnet sich folgendermaßen: $h_t = o_t \circ \sigma_h(c_t)$.

Varianten und Alternativen

Bevor LSTMs sich allgemein durchsetzten, wurden verzögerte Netze, sogenannte Time Delay Neural Networks, verwendet, später Hidden Markov Models.

Seit ihren Anfängen kamen immer mehr Varianten des LSTM hinzu. Wie oben beschrieben wurde zusätzlich das Forget Gate und die Peepholetechnik entwickelt sowie die Faltungstechnik. LSTM-Netze werden insbesondere in der Spracherkennung für die Klassifikation von Phonemen eingesetzt. Die erste Arbeit, die sich mit der Klassifikation von Phonemen mittels LSTM befasst, wurde 2005 von Alex Graves veröffentlicht[6]. 2010 wurde LSTM erstmals in einer Veröffentlichung von Martin Wöllmer für die Erkennung kontinuierlicher Sprache eingesetzt[7]. Forscher wie Haşim Sak[8] und Wojciech Zaremba[9] arbeiteten LSTM-Techniken für die akustische Modellierung und die Spracherkennung weiter aus.

Als Alternative zu LSTM wurden 2014 von Kyunghyun Cho und seinem Team *Gated Recurrent Units* entwickelt.[10] Diese werden besonders bei der Musikmodellierung eingesetzt. Sie kombinieren das Forget Gate und das Input Gate zu einem einzigen *Update Gate*. Das resultierende Modell ist einfacher als herkömmliche LSTM-Modelle und die Gates werden auf eine andere Art angeordnet.

Erfolge

In den Jahren nach 2010 verbesserte sich die technische Situation für LSTM außerordentlich: Die Einführung von Big Data stellte riesige Mengen von Daten zum Trainieren der Netze zu Verfügung. Der Boom von Computerspielen, bei denen die Akteure durch den Raum gehen, führte zu immer besseren und günstigeren Grafikkarten. Auf diesen Grafikkarten werden für die simulierte Bewegung der Akteure im Raum sehr viele Matrixmultiplikationen durchgeführt. Genau das braucht man für KI und LSTM. Schnelle GPU-Implementierungen dieser Kombination wurden 2011 durch Dan Ciresan und Kollegen in Schmidhubers Gruppe eingeführt.[11] Sie gewannen seither zahlreiche Wettbewerbe, u. a. die „ISBI 2012 Segmentation of Neuronal Structures in Electron Microscopy Stacks Challenge“[12] und den „ICPR 2012 Contest on Mitosis Detection in Breast Cancer Histological Images“.[13] Google entwickelte alternativ zum Grafikprozessor Tensor Processing Units, um Anwendungen im Rahmen von maschinellem Lernen zu beschleunigen. Sie werden unter anderem angewendet, um effektiv LSTMs zu verarbeiten.

Seit etwa 2016 setzen große Technologieunternehmen wie Google, Apple und Microsoft LSTM als grundlegende Komponente für neue Produkte ein. So verwendete Google beispielsweise LSTM für die Spracherkennung auf dem Smartphone[14], für den Smart Assistant Allo[15] und für Google Translate. Apple verwendet LSTM für die „Quicktype“-Funktion auf dem iPhone und für Siri[16]. Amazon verwendet LSTM für Amazon Alexa.[17]

Literatur

- Ramon Wartala: *Praxiseinstieg Deep Learning: Mit Python, Caffe, TensorFlow und Spark eigene Deep-Learning-Anwendungen erstellen*. Heidelberg 2018, ISBN 978-3960090540.

Weblinks

- Blog von Christopher Olah über LSTM (http://colah.github.io/posts/2015-08-Understanding-LSTMs/)
- Recurrent Neural Networks (http://www.idsia.ch/~juergen/rnn.html) mit über 30 LSTM Beiträgen von Jürgen Schmidhubers Team am IDSIA
- Olusola Adeniyi Abidogun: *Fraud detection paper...* (https://web.archive.org/web/20120522234026/http://etd.uwc.ac.za/usrfiles/modules/etd/docs/etd_init_3937_1174040706.pdf) mit zwei Kapiteln, die speziell LSTM behandeln.
- Lernhilfe, wie man LSTM in Python mit Theano einrichtet (http://christianherta.de/lehre/dataScience/machineLearning/neuralNetworks/LSTM.html)

Einzelnachweise

1. Sepp Hochreiter, Jürgen Schmidhuber: *Long short-term memory* In: Neural Computation (journal), vol. 9, issue 8, S. 1735–1780, 1997 online (https://www.researchgate.net/publication/13853244_Long_Short-term_Memory)
2. Sepp Hochreiter: *Untersuchungen zu dynamischen neuronalen Netzen* Diplomarbeit PDF (http://www.bioinf.jku.at/publications/older/3804.pdf) München 1991
3. Das Forget Gate wurde 2000 von Felix A. Gers und seinem Team entwickelt. Felix A. Gers, Jürgen Schmidhuber, Fred Cummins: *Learning to Forget: Continual Prediction with LSTM* In: Neural Computation (journal)vol. 12 issue 10, S. 2451–2471, 2000 online (http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.55.5709)
4. Felix Gers Dissertation (http://www.felixgers.de/papers/phd.pdf) über LSTM-Netze mit Forget Gate.
5. Xingjian Shi, Zhourong Chen, Hao Wang, Dit-Yan Yeung, Wai-kin Wong, Wang-chun Woo: *Convolutional LSTM Network: A Machine Learning Approach for Precipitation Nowcasting* In: *Proceedings of the 28th International Conference on Neural Information Processing Systems*, online (https://arxiv.org/abs/1506.04214), S. 802–810, 2015
6. Alex Graves, Jürgen Schmidhuber: *Framewise Phoneme Classification with Bidirectional LSTM Networks* In: Proc. of IJCNN 2005, Montreal, Canada, pp. 2047-2052, 2005 online (https://www.cs.toronto.edu/~graves/ijcnn_2005.pdf)
7. Martin Wöllmer, Florian Eyben, Björn Schuller, Gerhard Rigoll: *Recognition of Spontaneous Conversational Speech using Long Short-Term Memory Phoneme Predictions* In: Proc. of Interspeech 2010, ISCA, pp. 1946-1949, Makuhari, Japan, 2010 online (https://pdfs.semanticscholar.org/c7f7/2464f5925ecfca367a9679d30d82eb28eac8.pdf)
8. Haşim Sak, Andrew Senior, Françoise Beaufays: *Long Short-Term Memory Based Recurrent Neural Network Architectures for Large Vocabulary Speech Recognition* arxiv (https://arxiv.org/abs/1402.1128) 2014
9. Wojciech Zaremba, Ilya Sutskever, Oriol Vinyals: *Recurrent Neural Network Regularization* arxiv (https://arxiv.org/abs/1409.2329) 2014/2015
10. Cho, Kyunghyun; van Merrienboer, Bart; Gulcehre, Caglar; Bahdanau, Dzmitry; Bougares, Fethi; Schwenk, Holger; Bengio, Yoshua: *Learning Phrase Representations using RNN Encoder-Decoder for Statistical Machine Translation* arxiv (https://arxiv.org/abs/1406.1078) 2014.
11. Dan C. Ciresan, U. Meier, J. Masci, L. M. Gambardella, J. Schmidhuber: *Flexible, High Performance Convolutional Neural Networks for Image Classification.* (http://www.idsia.ch/~juergen/ijcai2011.pdf) International Joint Conference on Artificial Intelligence (IJCAI-2011, Barcelona), 2011.

12. Dan Ciresan, A. Giusti, L. Gambardella, J. Schmidhuber: *Deep Neural Networks Segment Neuronal Membranes in Electron Microscopy Images.* (http://www.idsia.ch/~juergen/nips2012.pdf) In: *Advances in Neural Information Processing Systems* (NIPS 2012), Lake Tahoe, 2012.
13. Dan Ciresan, A. Giusti, L. Gambardella, J. Schmidhuber: *Mitosis Detection in Breast Cancer Histology Images using Deep Neural Networks.* (http://www.idsia.ch/~juergen/miccai2013.pdf) MICCAI 2013.
14. Françoise Beaufays: *The neural networks behind Google Voice transcription (http://googleresearch.blogspot.co.at/2015/08/the-neural-networks-behind-google-voice.html)* (en-US). In: *Research Blog*, 11. August 2015. Abgerufen am 27. Juni 2017.
15. Pranav Khaitan: *Chat Smarter with Allo (http://googleresearch.blogspot.co.at/2016/05/chat-smarter-with-allo.html)* (en-US). In: *Research Blog*, 18. Mai 2016. Abgerufen am 27. Juni 2017.
16. Amir Efrati: *Apple's Machines Can Learn Too (https://www.theinformation.com/apples-machines-can-learn-too)*. 13. Juni 2016. Abgerufen am 27. Juni 2017.
17. Werner Vogels: *Bringing the Magic of Amazon AI and Alexa to Apps on AWS. - All Things Distributed (http://www.allthingsdistributed.com/2016/11/amazon-ai-and-alexa-for-all-aws-apps.html)*. 30. November 2016. Abgerufen am 27. Juni 2017.
