Hochschule für Angewandte Wissenschaften Hamburg
Hamburg University of Applied Sciences

Bachelorarbeit

Marina Knabbe

Titel

Fakultät Technik und Informatik
Studiendepartment Informatik

Faculty of Engineering and Computer Science
Department of Computer Science

Marina Knabbe

Titel

Bachelorarbeit eingereicht im Rahmen der Bachelorprüfung

im Studiengang Bachelor of Science Angewandte Informatik
am Department Informatik
der Fakultät Technik und Informatik
der Hochschule für Angewandte Wissenschaften Hamburg

Betreuender Prüfer: Prof. Dr. ErstprÃ¼fer
Zweitgutachter: Prof. Dr. ZweitprÃ¼fer

Eingereicht am: 1. Januar 2345

Marina Knabbe

Thema der Arbeit
Titel

Stichworte
SchlÃ¼sselwort 1, SchlÃ¼sselwort 2

Kurzzusammenfassung
Dieses Dokument …
Marina Knabbe

Title of the paper
English title

Keywords
keyword 1, keyword 2

Abstract
This document …

Inhaltsverzeichnis

Abbildungsverzeichnis

Listings

1 Einleitung (unfinished)

...

2 Künstliche neuronale Netze (unfinished)

- Ablauf: Training, Testing, Using?
- künstliche Intelligenz und Maschinen lernen
- Was können sie: label und prediction

http://deeplearning4j.org/neuralnet-overview

Neural networks are a set of algorithms, modeled loosely after the human brain, that are designed to recognize patterns. They interpret sensory data through a kind of machine perception, labeling or clustering raw input. The patterns they recognize are numerical, contained in vectors, into which all real-world data, be it images, sound, text or time series, must be translated. They help group unlabeled data according by similarities among the example inputs, and they classify data when they have a labeled dataset to train on. To be more precise, neural networks extract features that are fed to other algorithms for clustering and classification.

2.1 Feedforward Netzwerke (unfinished)

Ein Feedforward Netzwerk besteht aus Layern und Neuronen. Die Neuronen sind für die Berechnungen zuständig während die Layer den Aufbau des Netzes bestimmen. Abbildung 2.1 zeigt einen möglichen Aufbau eines künstliches Neurons.

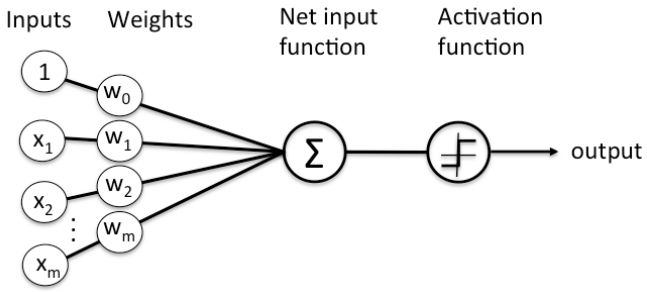


Abb. 2.1: mögliches Aussehen eines Neurons[1]

[1] Quelle: DL4J http://deeplearning4j.org/neuralnet-overview

Dieses Neuron besteht aus 1 bis x_m Eingängen (Inputs) mit Gewichten (Weights), einer Input Funktion (Net input function), einer Aktivierungsfunktion (Activation function) und einem Ausgang (Outputs). Die zu verarbeiteten Daten werden an die Eingänge gelegt, durch die zugehörigen Gewichte verstärkt oder abgeschwächt und anschließend aufsummiert. Die entstandene Summe wird dann an die Aktivierungsfunktion übergeben, welche das Ergebnis dieses Neurons festlegt.

Ein Layer besteht aus einer Reihe von Neuronen beliebiger Anzahl. Ein künstliches neuronales Netz setze sich aus einem Input Layer, einem Output Layer und beliebig vielen Hidden Layern zusammen. Hat ein Netz mehr als ein Hidden Layer so wird es auch als Deep Learning Netz bezeichnet.

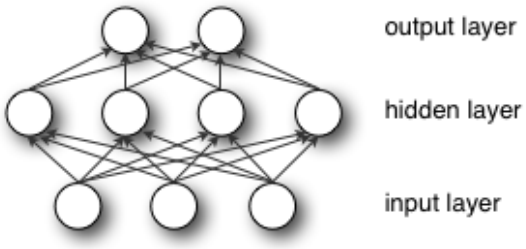


Abb. 2.2: Aufbau eines Feedforward Netzes[2]

Abbilding 2.2 zeigt ein Feedforward Netzwerk. Bei diesem Netz besitzt das Input Layer drei Neuronen, das Hidden Layer hat vier und das Output Layer hat zwei Neuronen. Die Ergebnisse des Input und Hidden Layers dienen dem nachfolgenden Layer als Eingang.

Ein neuronales Netz kann anhand von Trainingsdaten eine Funktion erlernen, indem es die Gewichte verändert. Da bei Trainingsdaten das Ergebnis bekannt ist, muss das Netz die Gewichte so bestimmen, dass sie mit den Eingangsdaten den gewünschten Ausgang mit möglichst kleinem Fehler abbilden. Das Ziel ist möglichst schnell den Punkt zu erreichen an dem der Fehler am kleinsten ist. Um dies zu erreichen wiederholt das Netz die folgenden Schritte: Ergebnis anhand der aktuellen Gewichte bestimmen, Fehler messen, Gewichte aktuallisieren. Eine weitverbreitete Optimierungsfunktions zur Gewichtebestimmung heißt Gradient Descent. Sie beschreibt das Verhältnis des Fehlers zu einem einzelnen Gewicht und wie sich der Fehler verändert wenn das Gewicht angepasst wird.

Sources:
- http://deeplearning4j.org/neuralnet-overview

[2] Quelle: DL4J http://deeplearning4j.org/neuralnet-overview

2.2 Der Vorteil von Deep Learning (unfinished)

- http://deeplearning4j.org/neuralnet-overview
In deep-learning networks, each layer of nodes trains on a distinct set of features based on the previous layerâ€™s output. The further you advance into the neural net, the more complex the features your nodes can recognize, since they aggregate and recombine features from the previous layer.

This is known as feature hierarchy, and it is a hierarchy of increasing complexity and abstraction. It makes deep-learning networks capable of handling very large, high-dimensional data sets with billions of parameters that pass through nonlinear functions.

Above all, these nets are capable of discovering latent structures within unlabeled, unstructured data, which is the vast majority of data in the world. Another word for unstructured data is raw media; i.e. pictures, texts, video and audio recordings. Therefore, one of the problems deep learning solves best is in processing and clustering the worldâ€™s raw, unlabeled media, discerning similarities and anomalies in data that no human has organized in a relational database or ever put a name to.

For example, deep learning can take a million images, and cluster them according to their similarities: cats in one corner, ice breakers in another, and in a third all the photos of your grandmother. This is the basis of so-called smart photo albums.

Now apply that same idea to other data types: Deep learning might cluster raw text such as emails or news articles. Emails full of angry complaints might cluster in one corner of the vector space, while satisfied customers, or spambot messages, might cluster in others. This is the basis of various messaging filters, and can be used in customer-relationship management (CRM). The same applies to voice messages. With time series, data might cluster around normal/healthy behavior and anomalous/dangerous behavior. If the time series data is being generated by a smart phone, it will provide insight into usersâ€™ health and habits; if it is being generated by an autopart, it might be used to prevent catastrophic breakdowns.

Deep-learning networks perform automatic feature extraction without human intervention

2.3 Recurrent Neuronal Networks (RNN) (unfinished)

http://deeplearning4j.org/lstm.html Recurrent networks, on the other hand, take as their input not just the current input example they see, but also what they perceived one step back in time. The decision a recurrent net reached at time step t-1 affects the decision it will reach one moment later at time step t. So recurrent networks have two sources of input, the present and the recent past, which combine to determine how they respond to new data, much

as we do in life. Recurrent networks are distinguished from feedforward networks by that feedback loop, ingesting their own outputs moment after moment as input. It is often said that recurrent networks have memory.2 Adding memory to neural networks has a purpose: There is information in the sequence itself, and recurrent nets use it to perform tasks that feedforward networks canâ€™t. That sequential information is preserved in the recurrent networkâ€™s hidden state, which manages to span many time steps as it cascades forward to affect the processing of each new example. Given a series of letters, a recurrent will use the first character to help determine its perception of the second character, such that an initial q might lead it to infer that the next letter will be u, while an initial t might lead it to infer that the next letter will be h.

https://colah.github.io/posts/2015-08-Understanding-LSTMs/ Humans donâ€™t start their thinking from scratch every second. As you read this essay, you understand each word based on your understanding of previous words. You donâ€™t throw everything away and start thinking from scratch again. Your thoughts have persistence.

Traditional neural networks canâ€™t do this, and it seems like a major shortcoming. For example, imagine you want to classify what kind of event is happening at every point in a movie. Itâ€™s unclear how a traditional neural network could use its reasoning about previous events in the film to inform later ones.

Recurrent neural networks address this issue. They are networks with loops in them, allowing information to persist. In the above diagram, a chunk of neural network, A, looks at some

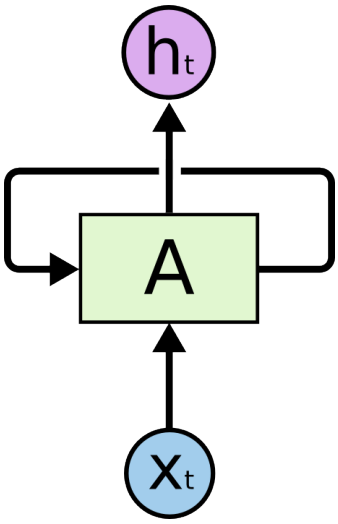


Abb. 2.3: RNN loop[3]

input xt and outputs a value ht. A loop allows information to be passed from one step of the network to the next.

These loops make recurrent neural networks seem kind of mysterious. However, if you think a bit more, it turns out that they arenâ€™t all that different than a normal neural network. A recurrent neural network can be thought of as multiple copies of the same network, each passing a message to a successor. Consider what happens if we unroll the loop: This chain-like

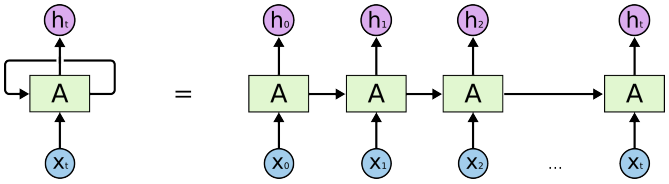


Abb. 2.4: unrolled RNN[4]

nature reveals that recurrent neural networks are intimately related to sequences and lists. Theyâ€™re the natural architecture of neural network to use for such data.

And they certainly are used! In the last few years, there have been incredible success applying RNNs to a variety of problems: speech recognition, language modeling, translation, image captioningâ€¦ The list goes on. Iâ€™ll leave discussion of the amazing feats one can achieve with RNNs to Andrej Karpathyâ€™s excellent blog post, The Unreasonable Effectiveness of Recurrent Neural Networks. But they really are pretty amazing.

Essential to these successes is the use of "LSTMs,ä very special kind of recurrent neural network which works, for many tasks, much much better than the standard version. Almost all exciting results based on recurrent neural networks are achieved with them. Itâ€™s these LSTMs that this essay will explore.

2.3.1 Backpropagation Through Time (BPTT) (unfinished)

http://deeplearning4j.org/lstm.html Remember, the purpose of recurrent nets is to accurately classify sequential input. We rely on the backpropagation of error and gradient descent to do so. Backpropagation in feedforward networks moves backward from the final error through the outputs, weights and inputs of each hidden layer, assigning those weights responsibility for a portion of the error by calculating their partial derivatives â€“ âˆ‚E/âˆ‚w, or the relationship between their rates of change. Those derivatives are then used by our learning rule, gradient descent, to adjust the weights up or down, whichever direction decreases error.

[3]Quelle: "colah"https://colah.github.io/posts/2015-08-Understanding-LSTMs/
[4]Quelle: "colah"https://colah.github.io/posts/2015-08-Understanding-LSTMs/

Recurrent networks rely on an extension of backpropagation called backpropagation through time, or BPTT. Time, in this case, is simply expressed by a well-defined, ordered series of calculations linking one time step to the next, which is all backpropagation needs to work.

Neural networks, whether they are recurrent or not, are simply nested composite functions like f(g(h(x))). Adding a time element only extends the series of functions for which we calculate derivatives with the chain rule.

Truncated BPTT

Truncated BPTT is an approximation of full BPTT that is preferred for long sequences, since full BPTTâ€™s forward/backward cost per parameter update becomes very high over many time steps. The downside is that the gradient can only flow back so far due to that truncation, so the network canâ€™t learn dependencies that are as long as in full BPTT.

2.3.2 Vanishing (and Exploding) Gradients (unfinished)

http://deeplearning4j.org/lstm.html Like most neural networks, recurrent nets are old. By the early 1990s, the vanishing gradient problem emerged as a major obstacle to recurrent net performance.

Just as a straight line expresses a change in x alongside a change in y, the gradient expresses the change in all weights with regard to the change in error. If we canâ€™t know the gradient, we canâ€™t adjust the weights in a direction that will decrease error, and our network ceases to learn.

Recurrent nets seeking to establish connections between a final output and events many time steps before were hobbled, because it is very difficult to know how much importance to accord to remote inputs. (Like great-great-*-grandparents, they multiply quickly in number and their legacy is often obscure.)

This is partially because the information flowing through neural nets passes through many stages of multiplication.

Everyone who has studied compound interest knows that any quantity multiplied frequently by an amount slightly greater than one can become immeasurably large (indeed, that simple mathematical truth underpins network effects and inevitable social inequalities). But its inverse, multiplying by a quantity less than one, is also true. Gamblers go bankrupt fast when they win just 97 cents on every dollar they put in the slots.

Because the layers and time steps of deep neural networks relate to each other through multiplication, derivatives are susceptible to vanishing or exploding.

Exploding gradients treat every weight as though it were the proverbial butterfly whose flapping wings cause a distant hurricane. Those weightsâ€™ gradients become saturated on

the high end; i.e. they are presumed to be too powerful. But exploding gradients can be solved relatively easily, because they can be truncated or squashed. Vanishing gradients can become too small for computers to work with or for networks to learn â€“ a harder problem to solve.

Below you see the effects of applying a sigmoid function over and over again. The data is flattened until, for large stretches, it has no detectable slope. This is analogous to a gradient vanishing as it passes through many layers.

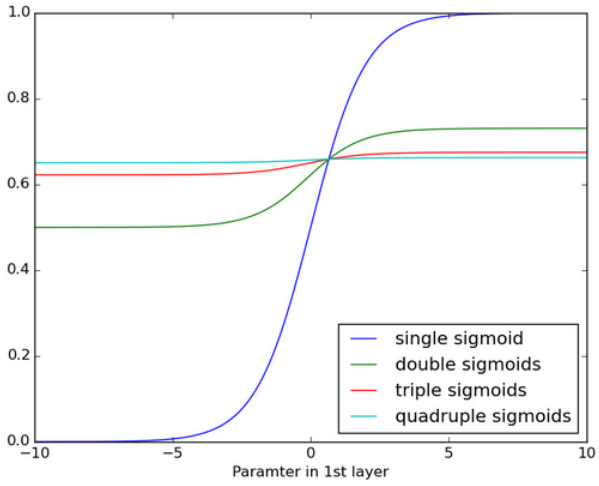


Abb. 2.5: Vanishing Gradient anhand von sigmoid function verdeutlicht[5]

2.3.3 The problem of long-term dependencies (unfinished)

One of the appeals of RNNs is the idea that they might be able to connect previous information to the present task, such as using previous video frames might inform the understanding of the present frame. If RNNs could do this, theyâ€™d be extremely useful. But can they? It depends.

Sometimes, we only need to look at recent information to perform the present task. For example, consider a language model trying to predict the next word based on the previous ones. If we are trying to predict the last word in ”the clouds are in the sky,”we donâ€™t need any further context â€“ itâ€™s pretty obvious the next word is going to be sky. In such cases, where the gap between the relevant information and the place that itâ€™s needed is small, RNNs can learn to use the past information. But there are also cases where we need more context. Consider trying to predict the last word in the text Ï grew up in Franceâ€¦ I speak fluent

[5]Quelle: DL4J http://deeplearning4j.org/lstm.htm
[6]Quelle: ”colah”https://colah.github.io/posts/2015-08-Understanding-LSTMs/

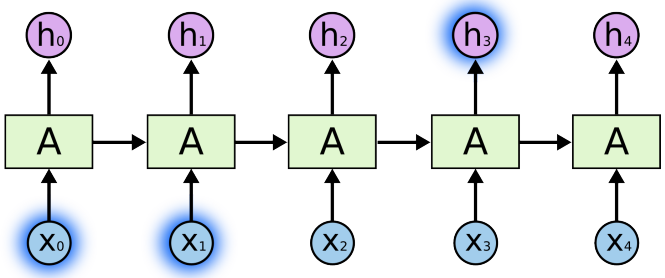


Abb. 2.6: RNN short term dependencies[6]

French.”Recent information suggests that the next word is probably the name of a language, but if we want to narrow down which language, we need the context of France, from further back. Itâ€™s entirely possible for the gap between the relevant information and the point where it is needed to become very large.

Unfortunately, as that gap grows, RNNs become unable to learn to connect the information. In theory, RNNs are absolutely capable of handling such ”long-term dependencies.Ä human

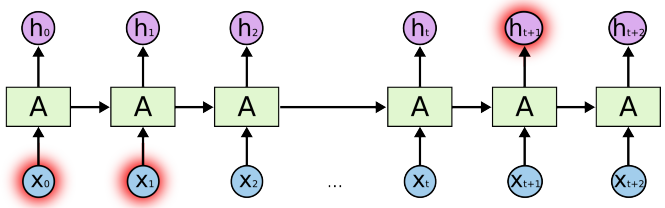


Abb. 2.7: RNN long term dependencies[7]

could carefully pick parameters for them to solve toy problems of this form. Sadly, in practice, RNNs donâ€™t seem to be able to learn them. The problem was explored in depth by Hochreiter (1991) [German] and Bengio, et al. (1994), who found some pretty fundamental reasons why it might be difficult.

Thankfully, LSTMs donâ€™t have this problem!

[7]Quelle: ”colah”https://colah.github.io/posts/2015-08-Understanding-LSTMs/

2.4 Long Short-Term Memory (LSTM) (unfinished)

http://deeplearning4j.org/lstm.html In the mid-90s, a variation of recurrent net with so-called Long Short-Term Memory units, or LSTMs, was proposed by the German researchers Sepp Hochreiter and Juergen Schmidhuber as a solution to the vanishing gradient problem.

LSTMs help preserve the error that can be backpropagated through time and layers. By maintaining a more constant error, they allow recurrent nets to continue to learn over many time steps (over 1000), thereby opening a channel to link causes and effects remotely.

LSTMs contain information outside the normal flow of the recurrent network in a gated cell. Information can be stored in, written to, or read from a cell, much like data in a computerâ€™s memory. The cell makes decisions about what to store, and when to allow reads, writes and erasures, via gates that open and close. Unlike the digital storage on computers, however, these gates are analog, implemented with element-wise multiplication by sigmoids, which are all in the range of 0-1. Analog has the advantage over digital of being differentiable, and therefore suitable for backpropagation.

Those gates act on the signals they receive, and similar to the neural networkâ€™s nodes, they block or pass on information based on its strength and import, which they filter with their own sets of weights. Those weights, like the weights that modulate input and hidden states, are adjusted via the recurrent networks learning process. That is, the cells learn when to allow data to enter, leave or be deleted through the iterative process of making guesses, backpropagating error, and adjusting weights via gradient descent.

The diagram below illustrates how data flows through a memory cell and is controlled by its gates.

Starting from the bottom, the triple arrows show where information flows into the cell at multiple points. That combination of present input and past cell state is fed not only to the cell itself, but also to each of its three gates, which will decide how the input will be handled.

The black dots are the gates themselves, which determine respectively whether to let new input in, erase the present cell state, and/or let that state impact the networkâ€™s output at the present time step. S_c is the current state of the memory cell, and g_y_in is the current input to it. Remember that each gate can be open or shut, and they will recombine their open and shut states at each step. The cell can forget its state, or not; be written to, or not; and be read from, or not, at each time step, and those flows are represented here.

The large bold letters give us the result of each operation.

[8] Quelle: DL4J http://deeplearning4j.org/lstm.htm

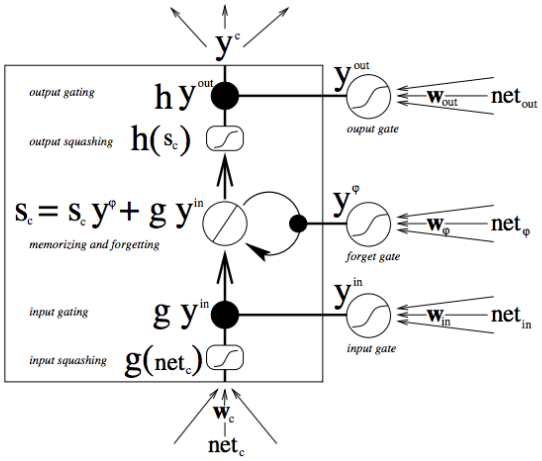


Abb. 2.8: Memory cell[8]

Hereâ€™s another diagram for good measure, comparing a simple recurrent network (left) to an LSTM cell (right). The blue lines can be ignored; the legend is helpful.

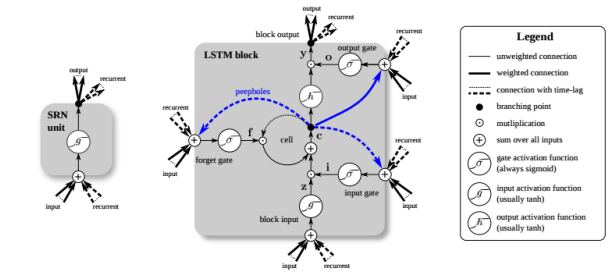


Figure 1. Detailed schematic of the Simple Recurrent Network (SRN) unit (left) and a Long Short-Term Memory block (right) as used in the hidden layers of a recurrent neural network.

Abb. 2.9: Vergleich: Simple Recurrent Network(SRN) und LSTM block[9]

Itâ€™s important to note that LSTMsâ€™ memory cells give different roles to addition and multiplication in the transformation of input. The central plus sign in both diagrams is essentially the secret of LSTMs. Stupidly simple as it may seem, this basic change helps them preserve a constant error when it must be backpropagated at depth. Instead of determining the subsequent cell state by multiplying its current state with new input, they add the two, and that quite literally makes the difference. (The forget gate still relies on multiplication, of course.)

[9]Quelle: DL4J http://deeplearning4j.org/lstm.htm

Different sets of weights filter the input for input, output and forgetting. The forget gate is represented as a linear identity function, because if the gate is open, the current state of the memory cell is simply multiplied by one, to propagate forward one more time step.

Furthermore, while weâ€™re on the topic of simple hacks, including a bias of 1 to the forget gate of every LSTM cell is also shown to improve performance.

You may wonder why LSTMs have a forget gate when their purpose is to link distant occurrences to a final output. Well, sometimes itâ€™s good to forget. If youâ€™re analyzing a text corpus and come to the end of a document, for example, you may have no reason to believe that the next document has any relationship to it whatsoever, and therefore the memory cell should be set to zero before the net ingests the first element of the next document.

In the diagram below, you can see the gates at work, with straight lines representing closed gates, and blank circles representing open ones. The lines and circles running horizontal down the hidden layer are the forget gates.

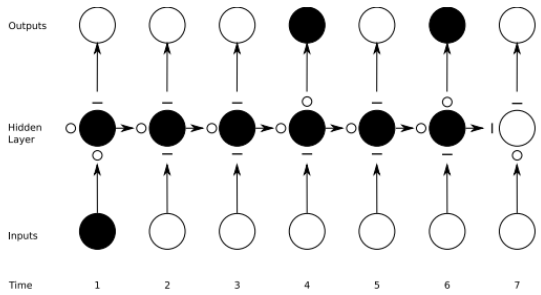


Abb. 2.10: Gates at work[10]

It should be noted that while feedforward networks map one input to one output, recurrent nets can map one to many, as above (one image to many words in a caption), many to many (translation), or many to one (classifying a voice).

”Capturing Diverse Time Scales and Remote Dependencies”

You may also wonder what the precise value is of input gates that protect a memory cell from new data coming in, and output gates that prevent it from affecting certain outputs of the RNN. You can think of LSTMs as allowing a neural network to operate on different scales of time at once.

[10]Quelle: DL4J http://deeplearning4j.org/lstm.htm

Letâ€™s take a human life, and imagine that we are receiving various streams of data about that life in a time series. Geolocation at each time step is pretty important for the next time step, so that scale of time is always open to the latest information.

Perhaps this human is a diligent citizen who votes every couple years. On democratic time, we would want to pay special attention to what they do around elections, before they return to making a living, and away from larger issues. We would not want to let the constant noise of geolocation affect our political analysis.

If this human is also a diligent daughter, then maybe we can construct a familial time that learns patterns in phone calls which take place regularly every Sunday and spike annually around the holidays. Little to do with political cycles or geolocation.

Other data is like that. Music is polyrhythmic. Text contains recurrent themes at varying intervals. Stock markets and economies experience jitters within longer waves. They operate simultaneously on different time scales that LSTMs can capture.

Further links: (could be sources for pictures) http://people.idsia.ch/ juergen/rnn.html https://karpathy.github.io/ effectiveness/ https://www.cs.cmu.edu/ bhiksha/courses/deeplearning/Fall.2015/pdfs/Werbos.backprop.pdf http://www.cs.toronto.edu/ graves/phd.pdf http://www.felixgers.de/papers/phd.pdf http://arxiv.org/pdf/1503.0406

https://colah.github.io/posts/2015-08-Understanding-LSTMs/ Long Short Term Memory networks â€“ usually just called ”LSTMs”â€“ are a special kind of RNN, capable of learning long-term dependencies. They were introduced by Hochreiter & Schmidhuber (1997), and were refined and popularized by many people in following work.1 They work tremendously well on a large variety of problems, and are now widely used.

LSTMs are explicitly designed to avoid the long-term dependency problem. Remembering information for long periods of time is practically their default behavior, not something they struggle to learn!

All recurrent neural networks have the form of a chain of repeating modules of neural network. In standard RNNs, this repeating module will have a very simple structure, such as a single tanh layer. LSTMs also have this chain like structure, but the repeating module has a

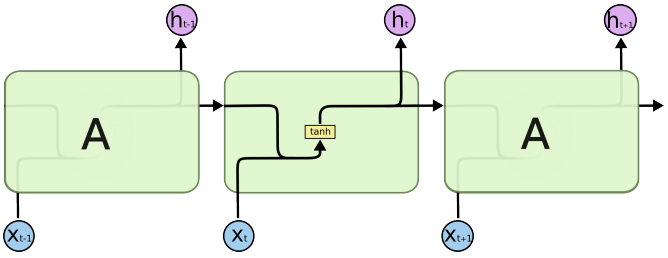


Abb. 2.11: The repeating module in a standard RNN contains a single layer.[11]

different structure. Instead of having a single neural network layer, there are four, interacting in a very special way. Donâ€™t worry about the details of whatâ€™s going on. Weâ€™ll walk

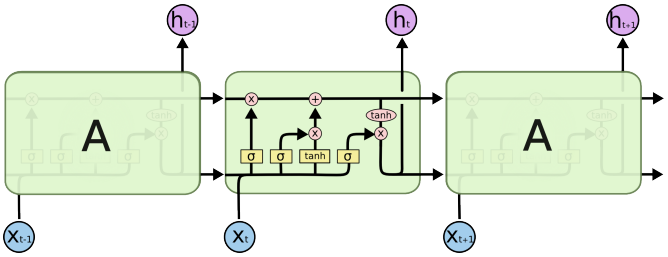


Abb. 2.12: The repeating module in an LSTM contains four interacting layers.[12]

through the LSTM diagram step by step later. For now, letâ€™s just try to get comfortable with the notation weâ€™ll be using. In the above diagram, each line carries an entire vector, from

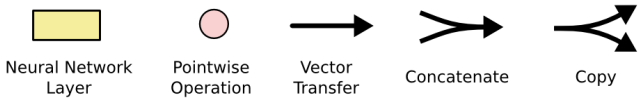


Abb. 2.13: used notation[13]

the output of one node to the inputs of others. The pink circles represent pointwise operations, like vector addition, while the yellow boxes are learned neural network layers. Lines merging denote concatenation, while a line forking denote its content being copied and the copies going to different locations.

The Core Idea Behind LSTMs

The key to LSTMs is the cell state, the horizontal line running through the top of the diagram. The cell state is kind of like a conveyor belt. It runs straight down the entire chain, with only some minor linear interactions. Itâ€™s very easy for information to just flow along it unchanged. The LSTM does have the ability to remove or add information to the cell state, carefully regulated by structures called gates.

Gates are a way to optionally let information through. They are composed out of a sigmoid neural net layer and a pointwise multiplication operation. The sigmoid layer outputs numbers

[11] Quelle: "colah" https://colah.github.io/posts/2015-08-Understanding-LSTMs/
[12] Quelle: "colah" https://colah.github.io/posts/2015-08-Understanding-LSTMs/
[13] Quelle: "colah" https://colah.github.io/posts/2015-08-Understanding-LSTMs/
[14] Quelle: "colah" https://colah.github.io/posts/2015-08-Understanding-LSTMs/

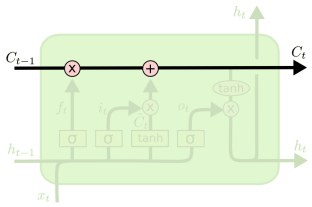


Abb. 2.14: LSTM C line[14]

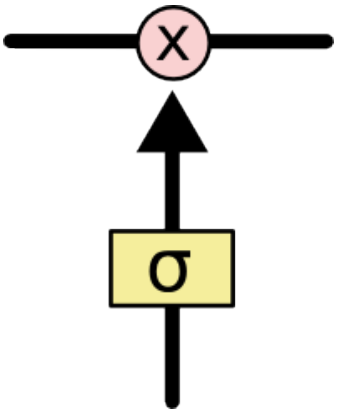


Abb. 2.15: LSTM gate[15]

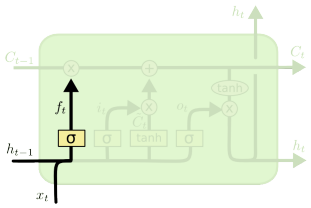
between zero and one, describing how much of each component should be let through. A value of zero means ”let nothing through,”while a value of one means ”let everything through!”

An LSTM has three of these gates, to protect and control the cell state.

Step-by-Step LSTM walk through

The first step in our LSTM is to decide what information weâ€™re going to throw away from the cell state. This decision is made by a sigmoid layer called the ”forget gate layer.Ït looks at htâˆ’1 and xt, and outputs a number between 0 and 1 for each number in the cell state Ctâˆ’1. A 1 represents ”completely keep this”while a 0 represents ”completely get rid of this.”

Letâ€™s go back to our example of a language model trying to predict the next word based on all the previous ones. In such a problem, the cell state might include the gender of the present subject, so that the correct pronouns can be used. When we see a new subject, we want to forget the gender of the old subject. The next step is to decide what new information



$$f_t = \sigma\left(W_f \cdot [h_{t-1}, x_t] \; + \; b_f\right)$$

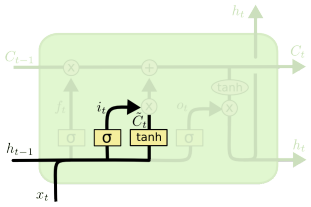
Abb. 2.16: LSTM focus f[16]

weâ€™re going to store in the cell state. This has two parts. First, a sigmoid layer called the ïnput gate layer”decides which values weâ€™ll update. Next, a tanh layer creates a vector of new candidate values, C t, that could be added to the state. In the next step, weâ€™ll combine these two to create an update to the state.

In the example of our language model, weâ€™d want to add the gender of the new subject to the cell state, to replace the old one weâ€™re forgetting. Itâ€™s now time to update the old cell state, Ctâˆ’1, into the new cell state Ct. The previous steps already decided what to do, we just need to actually do it.

We multiply the old state by ft, forgetting the things we decided to forget earlier. Then we add itâˆ—C t. This is the new candidate values, scaled by how much we decided to update each state value.

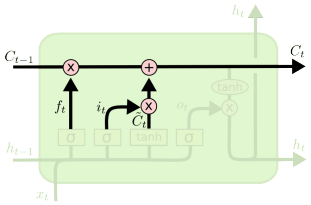
[15]Quelle: ”colah”https://colah.github.io/posts/2015-08-Understanding-LSTMs/
[16]Quelle: ”colah”https://colah.github.io/posts/2015-08-Understanding-LSTMs/
[17]Quelle: ”colah”https://colah.github.io/posts/2015-08-Understanding-LSTMs/



$$i_t = \sigma\left(W_i \cdot [h_{t-1}, x_t] \; + \; b_i\right)$$
$$\tilde{C}_t = \tanh(W_C \cdot [h_{t-1}, x_t] \; + \; b_C)$$

Abb. 2.17: LSTM focus i[17]

In the case of the language model, this is where weâ€™d actually drop the information about the old subjectâ€™s gender and add the new information, as we decided in the previous steps. Finally, we need to decide what weâ€™re going to output. This output will be based on our

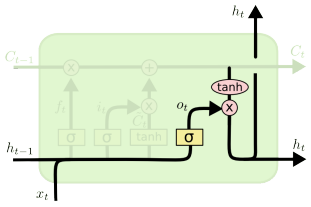


$$C_t = f_t * C_{t-1} + i_t * \tilde{C}_t$$

Abb. 2.18: LSTM focus C[18]

cell state, but will be a filtered version. First, we run a sigmoid layer which decides what parts of the cell state weâ€™re going to output. Then, we put the cell state through tanh (to push the values to be between â^'1 and 1) and multiply it by the output of the sigmoid gate, so that we only output the parts we decided to.

For the language model example, since it just saw a subject, it might want to output information relevant to a verb, in case thatâ€™s what is coming next. For example, it might output whether the subject is singular or plural, so that we know what form a verb should be conjugated into if thatâ€™s what follows next.



$$o_t = \sigma\left(W_o \; [h_{t-1}, x_t] \; + \; b_o\right)$$
$$h_t = o_t * \tanh\left(C_t\right)$$

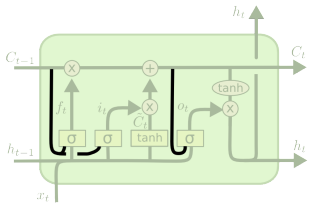
Abb. 2.19: LSTM focus o[19]

[18]Quelle: "colah"https://colah.github.io/posts/2015-08-Understanding-LSTMs/
[19]Quelle: "colah"https://colah.github.io/posts/2015-08-Understanding-LSTMs/

Variants on Long Short Term Memory

What Iâ€™ve described so far is a pretty normal LSTM. But not all LSTMs are the same as the above. In fact, it seems like almost every paper involving LSTMs uses a slightly different version. The differences are minor, but itâ€™s worth mentioning some of them.

One popular LSTM variant, introduced by Gers & Schmidhuber (2000), is adding "peephole connections."This means that we let the gate layers look at the cell state. The above diagram

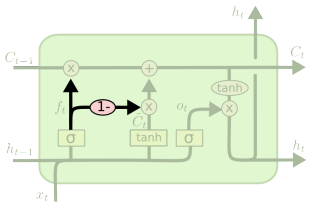


$$f_t = \sigma\left(W_f \cdot [\boldsymbol{C_{t-1}}, h_{t-1}, x_t] \; + \; b_f\right)$$
$$i_t = \sigma\left(W_i \cdot [\boldsymbol{C_{t-1}}, h_{t-1}, x_t] \; + \; b_i\right)$$
$$o_t = \sigma\left(W_o \cdot [\boldsymbol{C_t}, h_{t-1}, x_t] \; + \; b_o\right)$$

Abb. 2.20: LSTM peepholes[20]

adds peepholes to all the gates, but many papers will give some peepholes and not others.

Another variation is to use coupled forget and input gates. Instead of separately deciding what to forget and what we should add new information to, we make those decisions together. We only forget when weâ€™re going to input something in its place. We only input new values to the state when we forget something older. A slightly more dramatic variation on the LSTM



$$C_t = f_t * C_{t-1} + (1 - \boldsymbol{f_t}) * \tilde{C}_t$$

Abb. 2.21: LSTM tied[21]

is the Gated Recurrent Unit, or GRU, introduced by Cho, et al. (2014). It combines the forget and input gates into a single üpdate gate.Ït also merges the cell state and hidden state, and makes some other changes. The resulting model is simpler than standard LSTM models, and has been growing increasingly popular. These are only a few of the most notable LSTM variants. There are lots of others, like Depth Gated RNNs by Yao, et al. (2015). Thereâ€™s also some completely

[20] Quelle: "colah"https://colah.github.io/posts/2015-08-Understanding-LSTMs/
[21] Quelle: "colah"https://colah.github.io/posts/2015-08-Understanding-LSTMs/
[22] Quelle: "colah"https://colah.github.io/posts/2015-08-Understanding-LSTMs/

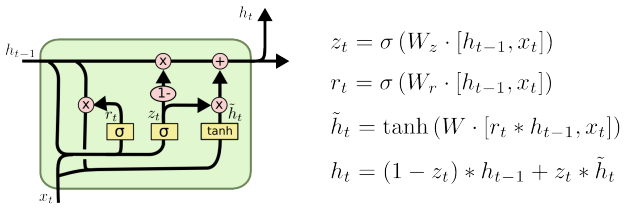


Abb. 2.22: LSTM GRU[22]

different approach to tackling long-term dependencies, like Clockwork RNNs by Koutnik, et al. (2014).

Which of these variants is best? Do the differences matter? Greff, et al. (2015) do a nice comparison of popular variants, finding that theyâ€™re all about the same. Jozefowicz, et al. (2015) tested more than ten thousand RNN architectures, finding some that worked better than LSTMs on certain tasks.

Conclusion

Earlier, I mentioned the remarkable results people are achieving with RNNs. Essentially all of these are achieved using LSTMs. They really work a lot better for most tasks!

Written down as a set of equations, LSTMs look pretty intimidating. Hopefully, walking through them step by step in this essay has made them a bit more approachable.

LSTMs were a big step in what we can accomplish with RNNs. Itâ€™s natural to wonder: is there another big step? A common opinion among researchers is: ”Yes! There is a next step and itâ€™s attention!”The idea is to let every step of an RNN pick information to look at from some larger collection of information. For example, if you are using an RNN to create a caption describing an image, it might pick a part of the image to look at for every word it outputs. In fact, Xu, et al. (2015) do exactly this â€“ it might be a fun starting point if you want to explore attention! Thereâ€™s been a number of really exciting results using attention, and it seems like a lot more are around the cornerâ€¦

Attention isnâ€™t the only exciting thread in RNN research. For example, Grid LSTMs by Kalchbrenner, et al. (2015) seem extremely promising. Work using RNNs in generative models â€“ such as Gregor, et al. (2015), Chung, et al. (2015), or Bayer & Osendorfer (2015) â€“ also seems very interesting. The last few years have been an exciting time for recurrent neural networks, and the coming ones promise to only be more so!

2.5 Aktueller Forschungsstand und Problemstellungen (unfinished)

Charakterisierungen ...

http://deeplearning4j.org/neuralnet-overview

As you think about one problem deep learning can solve, ask yourself: What categories do I care about? What information can I act upon? Those outcomes are labels that would be applied to data: spam or not_spam, good_guy or bad_guy, angry_customer or happy_customer. Then ask: Do I have the data to accompany those labels? Can I find labeled data, or can I create a labeled dataset (with a service like Mechanical Turk or Crowdflower) that I can use to teach an algorithm the correlation between labels and inputs?

http://deeplearning4j.org/lstm.html Recurrent nets are a type of artificial neural network designed to recognize patterns in sequences of data, such as text, genomes, handwriting, the spoken word, or numerical times series data emanating from sensors, stock markets and government agencies.

3 Deeplearning4j (unfinished)

See also DL4J
von http://deeplearning4j.org/quickstart
DL4J targets professional Java developers who are familiar with production deployments, IDEs and automated build tools.

How to: http://deeplearning4j.org/documentation ...

3.1 Getting started (unfinished)

von http://deeplearning4j.org/quickstart

3.1.1 Vorraussetzungen und Empfehlungen (unfinished)

- Java 1.7 oder hÃ¶her (nur 64-Bit Version wird unterstÃ¼tzt
- Apache Maven (Maven is a dependency management and automated build tool for Java projects.) check https://books.sonatype.com/mvnex-book/reference/public-book.html for how to use
- IntelliJ IDEA oder Eclipse (An Integrated Development Environment (IDE) allows you to work with our API and configure neural networks in a few steps. We strongly recommend using IntelliJ, which communicates with Maven to handle dependencies.)
- Git

Installation: http://deeplearning4j.org/gettingstarted Follow the ND4J Getting Started instructions to start a new project and include necessary POM dependencies. http://nd4j.org/getstarted.html http://nd4j.org/dependencies.htm

3.1.2 Ein neues Projekt in IntelliJ (unfinished)

http://nd4j.org/getstarted.html In IntelliJ - File -¿ new -¿ Project - GroupId(PackageName)

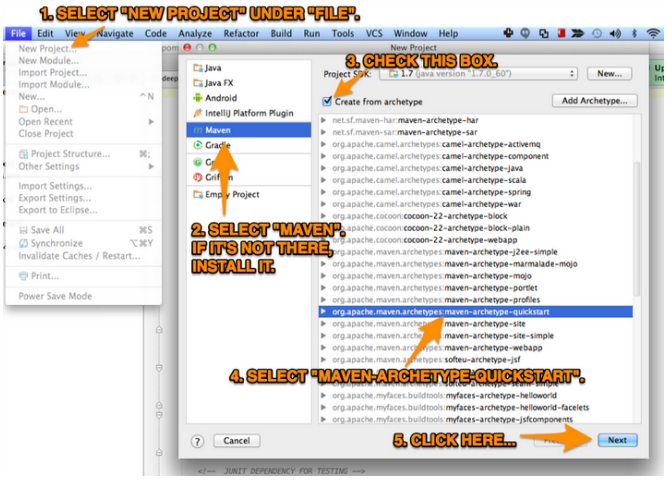


Abb. 3.1: Neues Maven Projekt erstellen[1]

und ArtifactId (Projektname?) festlegen - Finish

- gehe ins POM.xml file Update the POM file with the dependences youâ€™ll need. These will vary depending on whether youâ€™re running on CPUs or GPUs.

The default backend for CPUs is nd4j-native. You can paste that into the <dependencies> ... </dependencies> section of your POM like this:

```
<dependency>
  <groupId>org.nd4j</groupId>
  <artifactId>nd4j-native</artifactId>
  <version>${nd4j.version}</version>
</dependency>
```

Listing 3.1: applicationContext.xml

nd4j-cuda-7.5 (for GPUs)

ND4Jâ€™s version is a variable here. It will refer to another line higher in the POM, in the <properties> ... </properties> section, specifying the nd4j version and appearing similar to this:

```
  <nd4j.version>0.4-rc3.9</nd4j.version>
  <dl4j.version>0.4-rc3.10</dl4j.version>
```

Listing 3.2: applicationContext.xml

The DL4J dependencies you add to the POM will vary with the nature of your project.

[1]Quelle: ND4J http://nd4j.org/getstarted.html

In addition to the core dependency, given below, you may also want to install deeplearning4j-cli for the command-line interface, deeplearning4j-scaleout for running parallel on Hadoop or Spark, and others as needed.

```
    <dependency>
      <groupId>org.deeplearning4j</groupId>
      <artifactId>deeplearning4j-core</artifactId>
      <version>${dl4j.version}</version>
    </dependency>
```

Listing 3.3: applicationContext.xml

3.2 Feedforward Netze (unfinished)

http://deeplearning4j.org/quickstart.html Everything starts with a MultiLayerConfiguration, which organizes those layers and their hyperparameters. Hyperparameters are variables that determine how a neural network learns. They include how many times to update the weights of the model, how to initialize those weights, which activation function to attach to the nodes, which optimization algorithm to use, and how fast the model should learn. This is what one configuration would look like:

```
    MultiLayerConfiguration conf = new NeuralNetConfiguration.Builder()
        .iterations(1)
        .weightInit(WeightInit.XAVIER)
        .activation("relu")
        .optimizationAlgo(OptimizationAlgorithm.STOCHASTIC_GRADIENT_DESCENT)
        .learningRate(0.05)
        // ... other hyperparameters
        .backprop(true)
        .build();
```

With Deeplearning4j, you add a layer by calling layer on the NeuralNetConfiguration.Builder(), specifying its place in the order of layers (the zero-indexed layer below is the input layer), the number of input and output nodes, nIn and nOut, as well as the type: DenseLayer.

```
        .layer(0, new DenseLayer.Builder().nIn(784).nOut(250)
                .build())
```

Once youâ€™ve configured your net, you train the model with model.fit.

Configuring the POM.xml File

To run DL4J in your own projects, we highly recommend using Maven for Java users, or a tool such as SBT for Scala. The basic set of dependencies and their versions are shown below. This includes:

- deeplearning4j-core, which contains the neural network implementations - nd4j-native, the CPU version of the ND4J library that powers DL4J - canova-api - Canova is our library vectorizing and loading data

http://deeplearning4j.org/gettingstarted.html Reproducible Results

Neural net weights are initialized randomly, which means the model begins learning from a different position in the weight space each time, which may lead it to different local optima. Users seeking reproducible results will need to use the same random weights, which they must initialize before the model is created. They can reinitialize with the same random weight with this line:

```
1 Nd4j.getRandom().setSeed(123);
```

3.3 Recurrent Neuronal Network (unfinished)

3.3.1 Builder

http://deeplearning4j.org/doc/ Parametereinstellungen: - iterations(int): Number of optimization iterations. - learningRate(double): Learning rate. Defaults to 1e-1 - optimizationAlgo(OptimizationAlgorithm): - seed(long): Random number generator seed. Used for reproducability between runs - biasInit(double): ?? - miniBatch(boolean): Process input as minibatch vs full dataset. Default set to true. - updater(Updater): Gradient updater. - weightInit(WeightInit): Weight initialization scheme.

check http://deeplearning4j.org/glossary.html for explanations

- OptimizationAlgorithm: verfügbare Algorithmen - Updater: For example, Updater.SGD for standard stochastic gradient descent, Updater.NESTEROV for Nesterov momentum, Updater.RSMPROP for RMSProp, etc. (alle verfügbaren: ADADELTA, ADAGRAD, ADAM, CUSTOM, NESTEROVS, NONE, RMSPROP, SGD) - WeightInit: DISTRIBUTION Distribution: Sample weights from a distribution based on shape of input NORMALIZED Normalized: Normalize sample weights RELU Delving Deep into Rectifiers SIZE Size: Sample weights from bound uniform distribution using shape for min and max UNIFORM Uniform: Sample weights from

bound uniform distribution (specify min and max) VI VI: Sample weights from variance normalized initialization (Glorot) XAVIER N(0,2/nIn): He et al. (2015) ZERO Zeros: Generate weights as zeros

```
NeuralNetConfiguration.Builder builder = new NeuralNetConfiguration.
Builder();
builder.iterations(10);
builder.learningRate(0.001);
builder.optimizationAlgo(
OptimizationAlgorithm.STOCHASTIC_GRADIENT_DESCENT);
builder.seed(123);
builder.biasInit(0);
builder.miniBatch(false);
builder.updater(Updater.RMSPROP);
builder.weightInit(WeightInit.XAVIER);
```

Listing 3.4: Beispiel zur RNN-Erstellung: Builder

3.3.2 ListBuilder

first difference, for rnns we need to use GravesLSTM.Builder we need to use RnnOutputLayer for our RNN

// softmax normalizes the output neurons, the sum of all outputs is 1 // this is required for our sampleFromDistribution-function

```
ListBuilder listBuilder = builder.list(HIDDEN_LAYER_CONT + 1);

for (int i = 0; i < HIDDEN_LAYER_CONT; i++) {
    GravesLSTM.Builder hiddenLayerBuilder = new GravesLSTM.Builder();
    hiddenLayerBuilder.nIn(i == 0 ? LEARNSTRING_CHARS.size() :
HIDDEN_LAYER_WIDTH);
    hiddenLayerBuilder.nOut(HIDDEN_LAYER_WIDTH);
    hiddenLayerBuilder.activation("tanh");
    listBuilder.layer(i, hiddenLayerBuilder.build());
}

RnnOutputLayer.Builder outputLayerBuilder = new RnnOutputLayer.Builder(
LossFunction.MCXENT);
outputLayerBuilder.activation("softmax");
outputLayerBuilder.nIn(HIDDEN_LAYER_WIDTH);
outputLayerBuilder.nOut(LEARNSTRING_CHARS.size());
listBuilder.layer(HIDDEN_LAYER_CONT, outputLayerBuilder.build());

```

```
17      // finish builder
18      listBuilder.pretrain(false);
19      listBuilder.backprop(true);
20      listBuilder.build();
```

Listing 3.5: Beispiel zur RNN-Erstellung: ListBuilder

3.3.3 Netz erzeugen

```
1       // create network
2       MultiLayerConfiguration conf = listBuilder.build();
3       MultiLayerNetwork net = new MultiLayerNetwork(conf);
4       net.init();
5       net.setListeners(new ScoreIterationListener(1));
```

Listing 3.6: Beispiel zur RNN-Erstellung: Netz erzeugen

3.3.4 Daten erstellen

// create input and output arrays: SAMPLE_INDEX, INPUT_NEURON, // SEQUENCE_POSITION

```
1       INDArray input = Nd4j.zeros(1, LEARNSTRING_CHARS_LIST.size(),
        LEARNSTRING.length);
2       INDArray labels = Nd4j.zeros(1, LEARNSTRING_CHARS_LIST.size(),
        LEARNSTRING.length);
3       // loop through our sample-sentence
4       int samplePos = 0;
5       for (char currentChar : LEARNSTRING) {
6           // small hack: when currentChar is the last, take the first char as
7           // nextChar - not really required
8           char nextChar = LEARNSTRING[(samplePos + 1) % (LEARNSTRING.length)];
9           // input neuron for current-char is 1 at "samplePos"
10          input.putScalar(new int[] { 0, LEARNSTRING_CHARS_LIST.indexOf(
        currentChar), samplePos }, 1);
11          // output neuron for next-char is 1 at "samplePos"
12          labels.putScalar(new int[] { 0, LEARNSTRING_CHARS_LIST.indexOf(
        nextChar), samplePos }, 1);
13          samplePos++;
14      }
15      DataSet trainingData = new DataSet(input, labels);
```

Listing 3.7: Beispiel zur RNN-Erstellung: Daten erstellen

3.3.5 Netz trainieren

```
1     for (int epoch = 0; epoch < 100; epoch++) {
2         System.out.println("Epoch " + epoch);
3
4         // train the data
5         net.fit(trainingData);
6
7         // clear current stance from the last example
8         net.rnnClearPreviousState();
9
10        // put the first caracter into the rrn as an initialisation
11        INDArray testInit = Nd4j.zeros(LEARNSTRING_CHARS_LIST.size());
12        testInit.putScalar(LEARNSTRING_CHARS_LIST.indexOf(LEARNSTRING[0]),
      1);
13
14        // run one step -> IMPORTANT: rnnTimeStep() must be called, not
15        // output()
16        // the output shows what the net thinks what should come next
17        INDArray output = net.rnnTimeStep(testInit);
18
19        // now the net should guess LEARNSTRING.length mor characters
20        for (int j = 0; j < LEARNSTRING.length; j++) {
21            // first process the last output of the network to a concrete
22            // neuron, the neuron with the highest output cas the highest
23            // cance to get chosen
24            double[] outputProbDistribution = new double[LEARNSTRING_CHARS.
      size()];
25            for (int k = 0; k < outputProbDistribution.length; k++) {
26                outputProbDistribution[k] = output.getDouble(k);
27            }
28            int sampledCharacterIdx = findIndexOfHighestValue(
      outputProbDistribution);
29
30            // print the chosen output
31            System.out.print(LEARNSTRING_CHARS_LIST.get(sampledCharacterIdx)
      );
32
33            // use the last output as input
34            INDArray nextInput = Nd4j.zeros(LEARNSTRING_CHARS_LIST.size());
35            nextInput.putScalar(sampledCharacterIdx, 1);
36            output = net.rnnTimeStep(nextInput);
37        }
```

```
38            System.out.print("\n");
39        }
```

Listing 3.8: Beispiel zur RNN-Erstellung:

3.4 LSTM Netze (unfinished)

A commented example of a Graves LSTM learning how to replicate Shakespearian drama, and implemented with Deeplearning4j

Hyperparameter Tuning

http://deeplearning4j.org/lstm.html Here are a few ideas to keep in mind when manually optimizing hyperparameters for RNNs: - Watch out for overfitting, which happens when a neural network essentially ”memorizes”the training data. Overfitting means you get great performance on training data, but the networkâ€™s model is useless for out-of-sample prediction. - Regularization helps: regularization methods include l1, l2, and dropout among others. -So have a separate test set on which the network doesnâ€™t train. - The larger the network, the more powerful, but itâ€™s also easier to overfit. Donâ€™t want to try to learn a million parameters from 10,000 examples â€“ parameters ¿ examples = trouble. - More data is almost always better, because it helps fight overfitting. - Train over multiple epochs (complete passes through the dataset). - Evaluate test set performance at each epoch to know when to stop (early stopping). - The learning rate is the single most important hyperparameter. Tune this using deeplearning4j-ui; see [this graph] - In general, stacking layers can help. - For LSTMs, use the softsign (not softmax) activation function over tanh (itâ€™s faster and less prone to saturation (0 gradients)). - Updaters: RMSProp, AdaGrad or momentum (Nesterovs) are usually good choices. AdaGrad also decays the learning rate, which can help sometimes. - Finally, remember data normalization, MSE loss function + identity activation function for regression, Xavier weight initialization

4 ”Mein Beispiel”(unfinished)

...

5 Fazit (unfinished)

...

Literaturverzeichnis

[DL4J] : *Deeplearning4j: Open-source distributed deep learning for the JVM.* – URL http://deeplearning4j.org/. – Zugriffsdatum: 2016-06-30

Hiermit versichere ich, dass ich die vorliegende Arbeit ohne fremde Hilfe selbständig verfasst und nur die angegebenen Hilfsmittel benutzt habe.

Hamburg, 1. Januar 2345 Marina Knabbe