Rheinisch-Westfälische Technische Hochschule Aachen
Lehrstuhl für Informatik 6
Prof. Dr.-Ing. Hermann Ney

Character-based Embeddings of Words with Recurrent Nets (for Neural Language Modeling)

SS 2016

Character-based Embeddings of Words with Recurrent Nets

Article

Simon Grätzer

Matrikelnummer 311673

23.05.2016

Betreuer: Kazuki Irie

Contents

List of Tables

List of Figures

1 Introduction

When neural networks process image or audio-data they usually work with rich, high-dimensional datasets. For image data these might be vectors of individual pixel-intensities or for audio data vectors of power spectral density coefficients. Natural language processing systems traditionally treat words as atomic symbols which are encoded as simple indexes. For example 'apple' might become index $Id123$ and 'orange' becomes index $Id124$. This way the encoding of the word data is arbitrary and contains no meaningful information regarding the relationships between word-symbols anymore. In contrast to the dense representation of audio and image data, this way of encoding leads to data sparsity. Which means there is more data required to sufficiently train our statistical models.

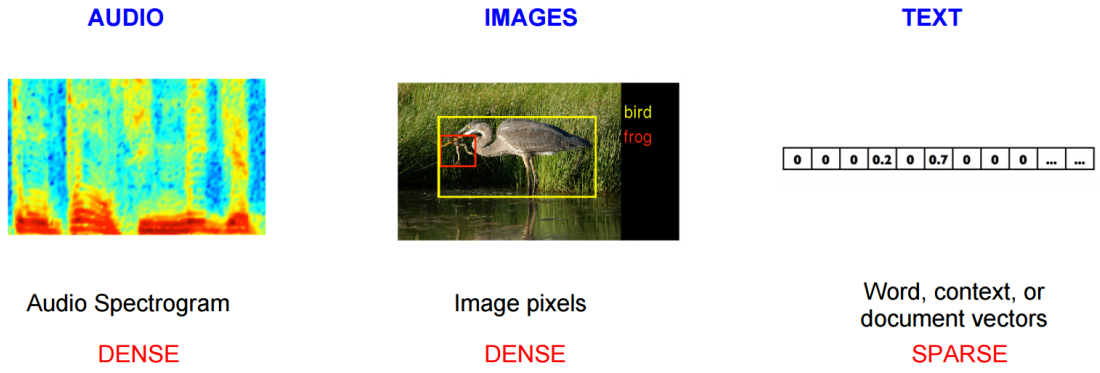

Figure 1: Comparison between the input datasets for different kinds of data (From [Inc, 2016]).

A statistical model can learn the relationships between word symbols (for example between 'apple' and 'apples') and leverage it for it's tasks. Essentially we need to have a function which can take a word from the vocabulary and turn it into a feature vector. The vectors are all in the same continuous vector space, in which we can now represent (embed) all words. Ideally these feature vectors have a low dimension compared to the size of the vocabulary. Additionally words which have a similar meaning (semantically similar) should be mapped to (geometrically) nearby points in the vector space. They capture the intuitive understanding that words may be different or similar along a variety of dimensions in a quantifiable way [Ling et al., 2015]. An example can be seen in figure 2, where some of these learned vectors representations capture a semantic relationship between different words.

These vector representations of words are called "word embeddings" and a natural language processing system can use different methods to generate them. All of these methods are based on the assumption that words which share semantic meaning tend to occur in the same contexts; this is called the "Distributional Hypothesis" [Sahlgren, 2008].

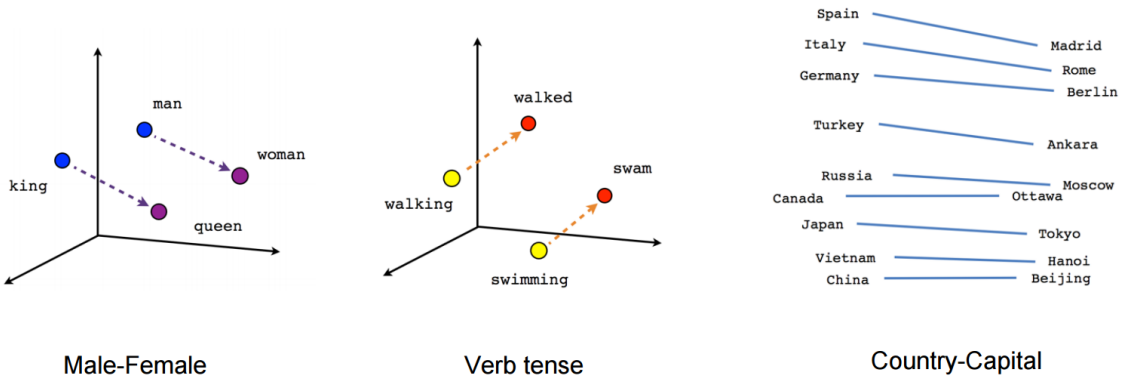

Figure 2: Visualization of word embeddings computed with the Skip-Gram model [Mikolov et al., 2013b]. Vectors are projected onto 2 dimensions (Figure taken from [Inc, 2016]).

Formally we have our vocabulary V and a function $f : V \to \mathbb{R}^d$ which maps $w \in V$ to a feature vector $\vec{(w)}$. A counting based approach to generate f would be to perform a dimensionality reduction on a word co-occurrence matrix [Lebret and Lebret, 2013]. The rows would then serve as the vectors.

However in this work we focus on recurrent neural networks to generate word embeddings. For example we will use the ability of neural nets to predict a word from the context (its neighbours) in which it appears. The first RNN will learn the vector embeddings as part of the larger processing system, in this case a neural natural language model (NNLM). By serving as the projection layer of the language model, the word embedding system is trained alongside with the rest of the model.

2 Recurrent Neural Networks

2.1 Introduction

Recurrent neural networks (RNN) are a class of neural networks where the connections in the graph can form directed cycles. This is basically similar to existing neuronal structures like biological brains. The cyclic nature of the RNN-graph allows the net to "remember" previous inputs over amounts of time, as opposed to other classes of neural networks which are directed acyclic graphs (also called "feed forward")

This has the advantage that the RNN can act based on previous events, without having to receive all the context as input at the same time. A simple example of an RNN can be seen in figure 3, where the net processes the input value x_i and outputs the value h_i. The way a loop in the net works is very simple: It's like copies of the same neural net chained up behind each other and passing it's current output value h_i to it's successor.

Following this idea the RNN can be "unrolled" to show the processing of different inputs x_i over time. An example of this can be seen in the figure 4. This way an RNN looks just like a feed forward neural net. This way of looking at recurrent connections will come up in the section 2.2 where we discuss the training of RNN's.

2.2 Training a RNN: Backpropagation Through Time

A common way to train neural networks is the so called "backpropagation of errors". It is a supervised learning method, this requires having a known desired output value for every training input. The measure of the difference between the desired and the actual output value of the net is the current error E, for example $E = \frac{1}{2}(h_i - x_i)^2$. The goal with backpropagation is to minimize the error E.

In other words we want to set every weight w_{ij} of the neural net such that E becomes minimal, the variable w_{ij} denotes the weight for a connection between neurons i and j. To minimize the error we can use the optimization method of "gradient descent". We calculate the gradient of E with regards to the weights of the neural net: $\frac{\partial E}{\partial w_{ij}}$ and then adjust every weight proportionally to the gradient:

$$\Delta w_{ij} = -\alpha \frac{\partial E}{\partial w_{ij}}$$

In this formula α is a factor to control the speed of the learning process and prevent overfitting

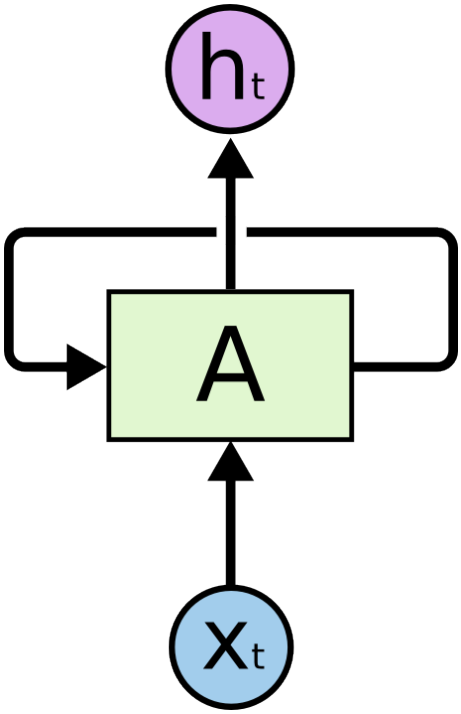

Figure 3: A basic Recurrent Neural Networks, taking in the value x_i and outputs the value h_i (taken from [Olah, 2015])

We can't just apply backpropagation to an RNN, because this doesn't account for the recursive nature of some of the links. A simple solution to this is the method of Backpropagation Through Time (BPTT). As previously described, the recursive link in the RNN can be "unrolled" such that the network becomes a normal feed forward network, see figure 4. The RNN loop is unrolled t times and now contains t instances of A with t inputs. If we have some sample data each training pattern consists of $\langle \mathbf{x}_0, \mathbf{y}_0, \mathbf{x}_1, \mathbf{x}_2, \ldots, \mathbf{x}_t, \mathbf{y}_t \rangle$, where $y_0, \ldots, y_t$ are the desired outputs. This sample can then be used for the training with backpropagation and gradient descend.

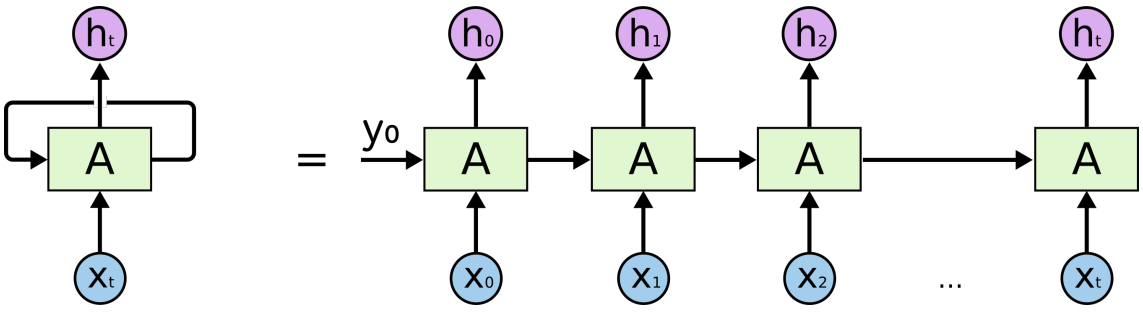

Figure 4: An unrolled recurrent neural network (taken from [Olah, 2015]).

2.2.1 Vanishing gradient problem

During training with backpropagation recurrent neural networks often suffer of the problem of vanishing gradients. When using BPTT for training a RNN the error gets backpropagated through potentially a great number of layers. The magnitude of the gradients of E is affected by the weights and the derivatives of the activation function. If either of these amounts to a factor of < 1 then the gradients can vanish over time, because we calculate the gradients with the chain rule $(f \circ g)' = (f' \circ g) \cdot g'$. This is a problem with a lot of common activation functions. For example the derivative of $\tanh(x)$ will be < 1 for all inputs $\neq 0$, the derivative of the sigmoid function is always $\frac{\partial sig(x)}{\partial x} \leq 0.25$. If a network has a lot of layers a lot of these small numbers get multiplied to calculate their gradient, and subsequently these layers train very slowly. If the gradients can become larger than 1 the gradients might become very big (exploding gradient problem).

There are a couple of solutions for this problem: With faster hardware the slow training of some layers may not be such a big issue anymore. Another solution is to avoid recurrent connections with gradients $\neq 1$. This way there are not many factors where the gradient can vanish. One such example is the LSTM network described in section 2.3. The activation function in the recurrency of the LSTM is the identity function, with a derivative of 1. Therefore the problems described above don't apply. There are other methods, but these are beyond our scope here.

2.3 Long-Short Term Memory

A special form of RNN which is designed to "remember" inputs over arbitary distances and "forget" them when necessary. They avoid the vanishing gradient problem are therefore very popular for tasks that involve any kind of classification, processing or prediction of time-series data. This ability makes them very well suited for tasks like neural language modeling or speech tagging where gap's between relevant inputs are likely. A basic LSTM block will have 3 essential gates:

- Gate i_t to determine when to learn an input value
- Gate f_t to determine if it should continue to remember or forget the currently stored value
- Gate o_t to determine wether it should output the value.

The figure 5 below shows a simple LSTM gate, with arrows showing the flow of the values. The gates with $\bigotimes$ calculate component-wise (Hadamard) product of their inputs. The activation function is usually the sigmoid function, except for the input/output gates marked with $\int$, which usually use the tanh function.

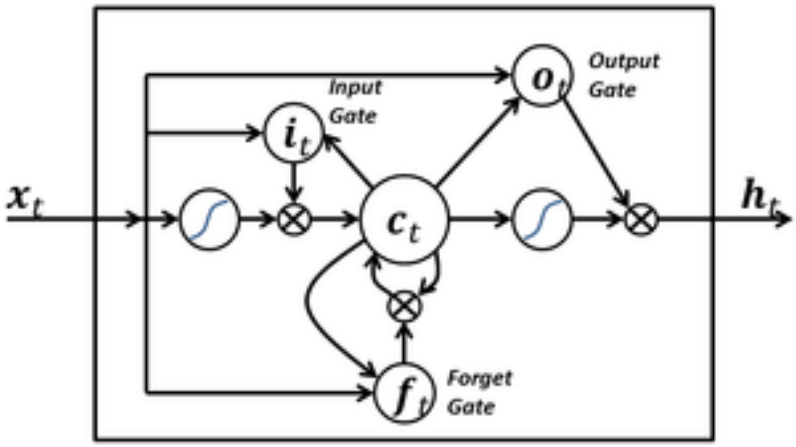

Figure 5: A LSTM gate with input, output, and forget gate.

Based on the input x_t and the current remembered value c_{t-1} the input gates i_t, f_t and o_t can decide how the LSTM behaves. If i_t is close to zero, no input from x_t is stored. If f_t is close to zero, the LSTM block will forget the stored value. Depending on o_t the block outputs the stored value or not. The output of c_t is not directly fed through an activation function, so it doesn't automatically decay over time.

Given the input vectors $x_1, \ldots, x_m$ a LSTM computes the output sequence $h_1, \ldots, h_{m+1}$ by iteratively applying the following equations (The W_* variables are the weight matrices of the gates):

$$\begin{aligned} i_t &= \sigma(W_{ix} * x_t + W_{ic} * c_{t-1}) \\ f_t &= \sigma(W_{fx} * x_t + W_{fc} * c_{t-1}) \\ o_t &= \sigma(W_{ox} * x_t + W_{oc} * c_t) \\ c_t &= f_t \odot c_{t-1} + i_t \odot \tanh(W_{cx} * x_t) \\ h_t &= o_t \odot \tanh(c_t) \end{aligned} \tag{1}$$

The sigmoid activation function is used for i_t, f_t, o_t since it's $(0, 1)$ range is suited well for their particular gating decision: These gates 'decide' how much of the value to pass through, which is a 0% to 100% decision. The tanh function is used as the output function, because it saturates at plus / minus one as opposed to zero and one for the sigmoid function [Glorot and Bengio, 2010].

As we will see, there are more complex variants of the LSTM. For example a possible addition is to feed the output value h_{t-1} into the gates i_t, f_t and o_t. This way they can incorporate the actual output into their gating decision.

2.4 Unsupervised training: Autoencoder

An autoencoder transforms an input sequence into a code, and vice versa. An autoencoder consists of two parts: the encoder and the decoder. They can be defined as transitions ϕ and ψ, such that:
$\phi : \mathcal{X} \to \mathcal{F}$
$\psi : \mathcal{F} \to \mathcal{X}$
$\arg\min_{\phi,\psi} \|X - (\psi \circ \phi)X\|^2$
This will allow us to train a NN to generate word embeddings without the need for labeled

data. Given a sequence of words this NN can learn a mapping where the resulting feature vectors are similar for words which have a similar meaning.

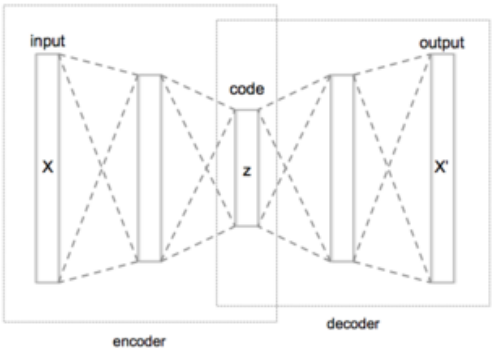

Figure 6: An autoencoder NN with the encoding and decoding layer

3 Introduction to Language Modeling

Language modelling is has the goal of determining a probability distribution on sequences of words (like sentences). A (statistical) language model estimates a function which calculates the probability p for the word sequence $w_1, \ldots, w_m$:

$$p(w_1, \ldots, w_m) = \prod_{i=1}^{m} p(w_i | w_1, \ldots, w_{i-1})$$

The fundamental challenge here is the dimensionality and vocabulary size of the data. For example to model the joint distribution of 10 word sequences with a vocabulary V size of 100.000 there are potentially 100000^10-1 free parameters [Yoshua Bengio and ChristianJanvin, 2003].

3.1 N-Gram Models

An n-gram is a contiguous sequence of n items / words from a given sequence of text or speech. Compared to the language model above, we only try to estimate a probability distribution for the last n words out of the sequence, we approximate the actual language model:

$$p(w_1, \ldots, w_m) = \prod_{i=1}^{m} p(w_i | w_1, \ldots, w_{i-1}) \approx \prod_{i=1}^{m} p(w_i | w_{i-(n-1)}, \ldots, w_{i-1})$$

So we assume we can approximate the probability for w_i by replacing the context of the preceding $i-1$ words with the context of the previous $n-1$ words. Formally this is a $n-1$ order Markov chain.

For a 1-gram sequence this is called an unigram, a 2-gram s a bigram, n = 3 is a trigram and so forth. One can estimate the conditional probabilities by counting all occurrences in the sample data:

$$P(w_i \mid w_{i-(n-1)}, \ldots, w_{i-1}) = \frac{\text{count}(w_{i-(n-1)}, \ldots, w_{i-1}, w_i)}{\text{count}(w_{i-(n-1)}, \ldots, w_{i-1})}$$

Of course this is very impractical, the number of possible sentences is too big. A training sample won't contain all possible combinations and therefore it's not possible to calculate estimations for all of them in this way.

3.1.1 Smoothing Techniques

We need to be able to estimate word combinations wich we have previously not seen and assign a non-zero probability to them. Some form of smoothing is necessary, the simplest possible solution is to just "add-one" to every word count. This is called "Laplacian smoothing" and allows us to assume a count of 1 for unseen n-gram's.

A better way is to use less context in these situations and to "Backoff". For example use a trigram instead of a 4-gram or a bigram and so on. Additionally it can be a good solutions to mix results from unigram, bigram, trigram etc and linearly interpolate between them. A more sophisticated way is to use neural networks, which represent words in a distributed way as a non-linear combinations of weights in a graph [Hinton et al., 1986].

3.2 Continuous space language models

Continuous space language models can base their predictions on continuous representations of words. The word embeddings, which we will discuss further in section 4, are a way to represent words in a continuous vector space. Continuous space embeddings help to alleviate the curse of dimensionality in language modeling, which arises out of the larger training texts. By representing the words in neural networks in a distributed way, we no longer have such a big issue with previously unseen word variations.

As with the simple counting based model from 3.1 the neural network continuous space language models train to estimate a probability distribution for the word w_i given the linguistic context.

$$p(w_i]\text{context})\forall w_i \in V$$

As with the n-gram models the context can be for example limited to a fixed number of previous words. In contrast to the simple counting model, a neural network can learn the complex non-local relationships between the word types in sequences. In section 5.1 we are going to discuss how to apply the word embeddings from section 4 in a neural natural language model.

4 Character-based Word-Embeddings

4.1 Introduction

The Compositional Characters to Word (C2W) Model is a new way to generate word embeddings presented by [Ling et al., 2015]. The input of the C2W model is a single word w, and we want to get a d-dimensional vector by which to represent w. The basic idea is to let two LSTMs 'read' the word, by feeding them the word as a sequence of characters forwards and backwards. The two resulting output states are then combined into the vector representation of the word.

A commonly previously used approach is to treat the word embeddings as optimizable parameters in some kind of language model. After training the resulting vector representations can then be reused in other tasks. Such models basically generate a lookup table in which the word vectors are stored. This has the advantage that tasks with low

amounts of training data can be trained together with tasks with larger amounts of data. The lookup table is formally justa matrix over the vocabulary $P \in mathbbR^{V \times d}$ where d is the dimension of the embedding. The embedding is obtained from P by multiplying with a selection vector 1_w, which is 1 at the word index: $e_w^W = P * 1_w$. For example the previously mentioned skip-gram model [Mikolov et al., 2013a] works in this way.

The previous approach has some drawbacks which the C2W seeks to avoid:

1. Each word embedding vector is independent. A word lookup table cannot generate representations for an unknown word, even though it might have seen similar words before. A model might capture the linear correspondence of *cat* and *apple* compared to *cats* and *apples*, but it doesn't capture that the added *s* is responsible for this transformation. It can't compute the same for a previously unseen plural word, even though it might know the singular version.

2. For a large vocabulary it becomes impractical to actually store all word embeddings in a table.

The C2W model avoids the first problem by breaking down words into their components. These components are then composed back into the the representation of the word. There are previous efforts ([Luong et al., 2013], [Botha and Blunsom, 2014]) which explore this idea, by breaking up words into their *morphemes*. A morpheme is the smallest grammatical unit of a language e.g. "Unbreakable" comprises three morphemes: un-, -break-, and -able. The downside of this approach is that you need a morphological analyzer to break down words. The C2W avoid this dependency by breaking down words directly into characters.

4.2 Character Lookup Table

The first processing steps involves a table of d_C parameters for each character from a predefined character alphabet C. Each word w is decomposed into it's characters $c_1, \ldots . c_m$. Every input character gets transformed into a d_C-dimensional feature vector $e_{c_j}^C$ from this table. These character feature vectors represent the properties of characters in the target language, for example consonants vs vowels. The feacture vectors are stored in a projection layer $P_C \in \mathbb{R}^{d_C \times |C|}$ (Which is basically a lookup table for characters). For each c_j we can define the projection of our character as $e_{c_j}^C = P_C * 1_{c_j}$. In this equation the $1_{c_j} \in 0, 1^{d_C}$ is a vector where all entries are zero except the entry corresponding to the index of the characterc_j. The resulting sequence of character representations $e_{c_1}^C, \ldots, e_{c_m}^C$ is then fed into the Bi-LSTM layer 4.3 in normal and reversed order.

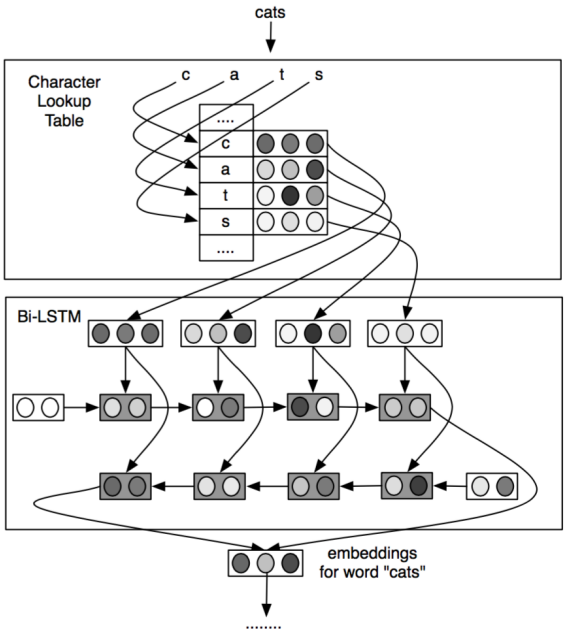

Figure 7: Illustration of the lexical Composition Model (from [Ling et al., 2015])

4.3 Bidirectional LSTM Layer

The second layer of the C2W model
is a bidirectional LSTM layer [Graves and Schmidhuber, 2005].
This consists of two LSTM blocks:

1. The forward LSTM (f) which receives the sequence of character representations $e^C_{c_1}, \ldots, e^C_{c_m}$

2. The backward LSTM (b) which receives as input the reverse sequence $e^C_{c_m}, \ldots, e^C_{c_1}$

The two LSTM blocks yield the forward state sequence $s^f_0, \ldots, s^f_m$ and backward state sequence $s^b_m, \ldots, s^b_0$. The reason for using a bidirectional LSTM is explained in a more detail in section 4.3.1. A shortened explanation is that by supplying the network with the reversed input, it additionally gains access to previous context (previous characters).

The LSTM blocks are a little extended in comparison to the version introduced in section 2.3. The main extension here is to feed the output value h_{t-1} back into the decision gates i_t, f_t and o_t. This way the gates can incorporate the actual output of the LSTM block into their gating decision. Futhermore there are bias values b_* added to all gates. The bias values can improve the performance of the LSTM compared to other RNN architectures [Jozefowicz et al., 2015] The equations therefore look a little different:

$$\begin{aligned} i_t &= \sigma(W_{ix} * x_t + W_{ih} * h_{t-1} + W_{ic} * c_{t-1} + b_i) \\ f_t &= \sigma(W_{fx} * x_t + W_{fh} * h_{t-1} + W_{fc} * c_{t-1} + b_f) \\ o_t &= \sigma(W_{ox} * x_t + W_{oh} * h_{t-1} + W_{oc} * c_t + b_o) \\ c_t &= f_t \odot c_{t-1} + i_t \odot \tanh(W_{cx} * x_t + W_{ch} * h_{t-1} + b_c) \\ h_t &= o_t \odot \tanh(c_t) \end{aligned} \tag{2}$$

There are two seperate parameter sets for the LSTM block $\mathcal{W} = \{W_{ix}, \ldots, W_{ch}, b_i, \ldots, b_o\}$:
One for the forward LSTM ($\mathcal{W}^f$) and one for the backward ($\mathcal{W}^b$) LSTM.

4.3.1 Bidirectional Recurrent Neural Networks

The idea of a bidirectional recurrent neural network (BRNN) is to present every training sequence forwards and backwards to two separate recurrent neural networks [Schuster and Paliwal, 1997]. Both networks are connected to the same output layer. This way the complete network has access to sequential information about all inputs before and after the current one. This is illustrated in figure 8, where the complete input is provided to the RNN twice (forwards and backwards).

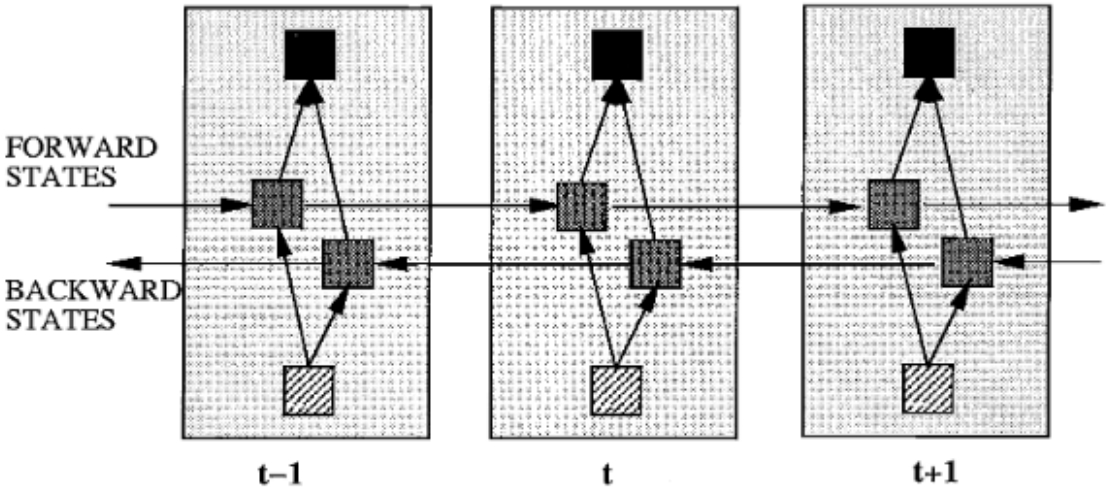

Figure 8: A generic BRNN unfolded for three timesteps (From [Schuster and Paliwal, 1997]).

The advantage of a BRNN is that there is no need to anymore manually specify how much context is used anymore (for example by having fixed-sized overlapping time-windows). The net can decide to use as much or as little of the context as required. This can improve the results in sequence learning tasks, where the context can be previded in a "semi-online" fashion. For example in online speech recognition an output after every sentences is fine [Graves and Schmidhuber, 2005].

4.4 Output-Layer

Finally the last states of the forward sequence s_m^f and the backwards sequence s_0^b are linearly combined into the resulting word embedding e_w^C:

$$e_w^C = D^f s_m^f + D^b s_0^b + b_d$$

The variables D^f, D^b, b_d are the weights which determine how the states are combined.

Compared to using a lookup table computing e_w^C is relatively expensive, but since the value only depends on the word itself the value can be cached. This can be used for frequently occuring words to reduce the computational load. At the same time not all words have to be cached, which reduces the amount of storage required for a large vocabulary.

5 Applying the C2W model

5.1 Language Modeling

In this section we use the C2W model and a recurrent LSTM to perform language modeling as described in [Ling et al., 2015]. The neural network is going to compute the log propability $\log p(\sqsupseteq)$ of the sentence $\sqsupseteq = (w_1, \ldots, w_m)$. As previously discussed in section 3 this can be decomposed into the sum of the conditional log probabilities $logp(w_1, \ldots, w_m) = \sum_{i=1}^{m} logp(w_i|w_1, \ldots, w_{i-1})$. This model will compose the previously

described word representations of $w_1, \ldots, w_{i-1}$ to compute $logp(w_i|w_1, \ldots, w_{i-1})$ using a LSTM unit. The model is shown in figure 9. The first stage can use either the embeddings generated with the C2W model $f(w_i) = e^C_{w_i}$ or simply word lookup tables $e^W_{w_i} = P * 1_{w_i}$.

Every time we input a new word w_i from the sequence we get the LSTM state s_i. The state s_i is then used to predict word w_{i+1}. The state s_i is projected into a vector of size $|V|$ (the vocabulary size). The softmax is a $d \times V$ table, which encodes the likelihood of every word type in the vocabulary in a given context. The softmax layer will ensure a valid log probability distribution. This way we can choose the word type with the maximal likelihood. Maximizing the conditional log-likelihoods during training will improve the modeling of $p(\sqsupseteq)$ [Mikolov et al., 2010]. More details regarding the softmax layer are explained in section 5.1.2.

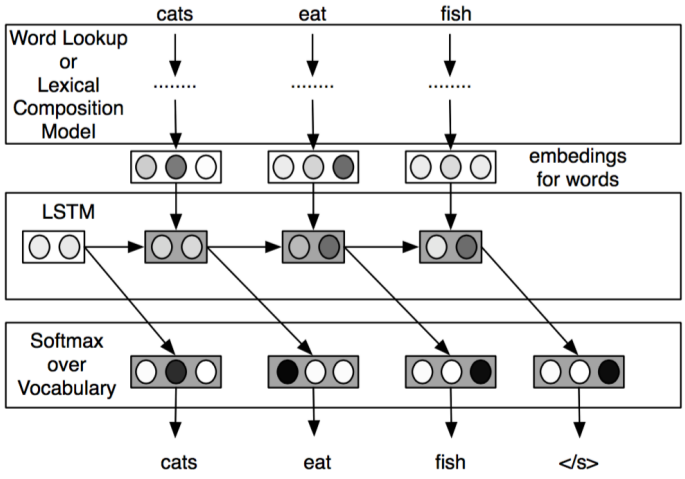

Figure 9: Neural network for language model. The C2W model is at the top, the word embeddings are then fed into an recurrent LSTM. Finally

5.1.1 Out of Vocabulary Token

Since the softmax can only be computed for known words included in the vocabulary set V it is not possible to handle words not included in V during testing. These words are **Out Of Vocabulary** (OOV), the solution is to replace all of them with an special token word. The probability for this OOV token can be learned during the training stage, by replacing word types with low frequencies with the ¡OOV¿ token. During test time we can now use the ¡OOV¿ probabilities for any word not in the vocabulary V [Ling et al., 2015].

For a language modeling task where there is not a closed Vocabulary, we will almost certainly encounter words which are not known from our training vocabulary set. It is therefore essential to apply a strategy like the above one to have a robust estimation for OOV words.

5.1.2 Softmax Layers

The Softmax function is a *logistics function* is often used as the final combining processing layer in neural networks, as discussed in this article. It calculates a normalized exponential

that 'squashes' the input such that the resulting vector entries add up to 1.

$$\sigma(\mathbf{z})_j = \frac{e^{z_j}}{\sum_{k=1}^{|V|} e^{z_k}}$$

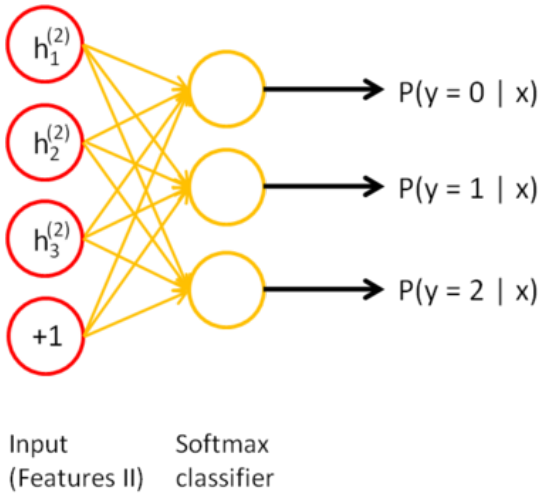

Figure 10: Structure of a softmax layer in a neural network

The property of the softmax layer is that the result of the outputs can be interpreted as posterior probabilities. This is useful in classification as it gives a valid posteriori propability measure for an n-gram model. A softmax layer will be used to transform the results of a LSTM layer into a probability distribution for our neural language models.

5.1.3 Perplexity

As we have previously seen, language modeling amounts to learning a posteriori probability distribution. Perplexity is a measure of how well a probability distribution predicts sample data. It can be interpreted as the number of choices per word position. We can use it to formally judge how well a language model can predict the distribution. The exponent in the Perplexity term below is the Entropy, it expresses the number of bits required to represent an event (in our case a word or character).

Formally Perplexity is defined as the sum of the inverse log-probabilities

$$2^{H(p)} = 2^{-\sum_x p(x) \log_2 p(x)}$$

Where $H(p)$ is the entropy [Shannon, 1948] of the probability distribution X of values $x_1, \ldots, x_n$ with the probability mass function $p : X \to [0, 1]$.

To minimize the perplexity means to have a better fitting language model. This is relevant for the language model discussed in the previous section, because the entropy is used in the orginal paper [Ling et al., 2015] to evaluate the performance of the language model with different configurations. However the perplexity should be used to compare models with different vocabularies. Since there are fewer outcomes in the softmax, the global perplexity can decrease.

5.1.4 Entropy

Shannon [Shannon, 1948] or Bit-Entropy is defined as $log_2(n)$ where n is the number of possible outcomes. This assumes that all possible outcomes have the same probability. For a probability distribution we need to consider all thre possible values $x_1, \ldots, x_n$ it's likelihood of appearance and the entropy every possible outcome has.

The entropy of a value x_i is dependent on it's self-information, which is inversely proportional to it's probability: $I(x_i) = log(\frac{1}{p(x_i)}) = -log(p(x_i))$. The self information of an intersection of two independent events A, B is the sum of both: $I(A \cap B) = I(A) + I(B)$. This is why the log function is used to wrap $\frac{1}{p(x_i)}$. In total the entropy for X is defined as the expected information content of X:

$$H(p) = E(I(X)) = -\sum_{i=1}^{n} p(x_i) \log_2 p(x_i)$$

5.2 Morphological inflection generation

Another application of C2W like model beyond language modeling is the morphological inflection generation [Faruqui et al., 2015]. The goal is to perform a lingustic transformation some examples of the inflections which this model seeks to generate can be seen in table 1. First stage is the **encoder**:
This part of the model is eqvivalent to the C2W model: Each word w is broken up into it's characters $e^C_{c_1}, \ldots, e^C_{c_m}$. Then a bidirectional LSTM layer is used and th result is composed into e_w. Second part is the **decoder**:
This is a LSTM unit which will sequentially receive the word characters as input combined with the previous output and the word vector. Every output state is computed as $s_t = g(s_{t-1}, \{e_w, y_{t-1}, x_t\})$. Where g is the decoder LSTM unit, s_t is the LSTM output, y_{t-1} the actual character output and x_t is the current input character. Since the input word might be shorter than the output, onece the input sequence runs out of characters we feed in an ϵ character indicating null input: $x_t = \epsilon$. The vector representation for the ϵ character is learned just like in the C2W model.

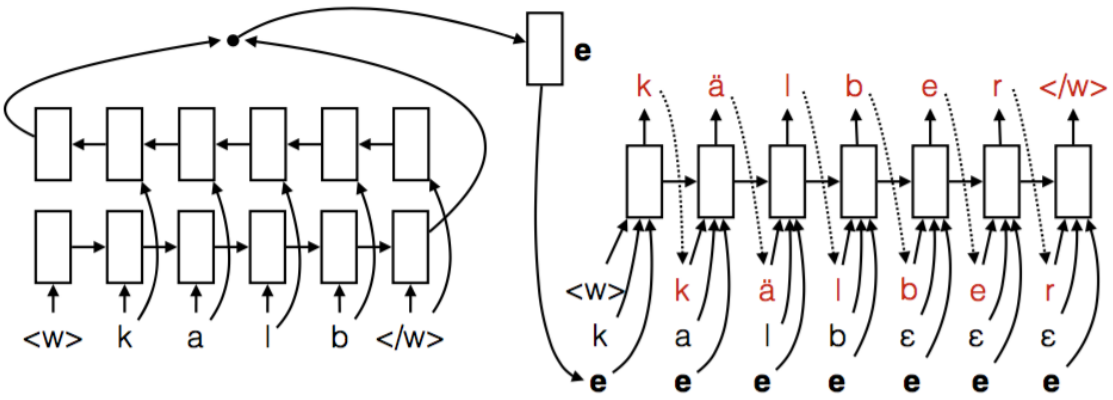

Figure 11: A endoder decoder structure for inflection generation

	singular	plural
nominative	Kalb	Kälber
accusative	Kalb	Kälber
dative	Kalb	Kälbern
genitive	Kalbes	Kälber

Table 1: Example of an inflection table for the word Kalb (calf, German).

Converting the word embeddings from a C2W like model back into inflected versions of the word as discussed in Faruqui et al. [Faruqui et al., 2015]

6 Conclusion

The model of composing word embeddings by composing characters shows that it is possible to avoid feature enginerring as well as costly lookup-tables. Using characters as the atomic unit of choice makes the model very simple to apply to many languages. As the experiments in [Ling et al., 2015] show, lexical features can can be learned automatically by the language model. Large word lookup-tables can be reduced to one (much smaller) character lookup table, without sacrificing performance. The model therefore avoids redundancies in word lookup tables. This is an additional advantage over morphenes as atomic units of words. The lookup table allows the model to scale better with larger amounts of data.

References

[Botha and Blunsom, 2014] Botha, J. A. and Blunsom, P. (2014). Compositional morphology for word representations and language modelling. *CoRR*, abs/1405.4273.

[Faruqui et al., 2015] Faruqui, M., Tsvetkov, Y., Neubig, G., and Dyer, C. (2015). Morphological inflection generation using character sequence to sequence learning. *CoRR*, abs/1512.06110.

[Glorot and Bengio, 2010] Glorot, X. and Bengio, Y. (2010). Understanding the difficulty of training deep feedforward neural networks. In *In Proceedings of the International Conference on Artificial Intelligence and Statistics (AISTATS10). Society for Artificial Intelligence and Statistics*, pages 249–256.

[Graves and Schmidhuber, 2005] Graves, A. and Schmidhuber, J. (2005). Framewise phoneme classification with bidirectional LSTM and other neural network architectures. *Neural Networks*, 18(5-6):602–610.

[Hinton et al., 1986] Hinton, G. E., McClelland, J. L., and Rumelhart, D. E. (1986). Distributed representations. In Rumelhart, D. E., McClelland, J. L., and PDP Research Group, C., editors, *Parallel Distributed Processing*, chapter 3, pages 77–109. MIT Press.

[Inc, 2016] Inc, G. (2016). Vector representations of words. [Online; accessed 22-May-2016].

[Jozefowicz et al., 2015] Jozefowicz, R., Zaremba, W., and Sutskever, I. (2015). An empirical exploration of recurrent network architectures. In Blei, D. and Bach, F., editors, *Proceedings of the 32nd International Conference on Machine Learning (ICML-15)*, pages 2342–2350. JMLR Workshop and Conference Proceedings.

[Lebret and Lebret, 2013] Lebret, R. and Lebret, R. (2013). Word emdeddings through hellinger PCA. *CoRR*, abs/1312.5542.

[Ling et al., 2015] Ling, W., Luís, T., Marujo, L., Astudillo, R. F., Amir, S., Dyer, C., Black, A. W., and Trancoso, I. (2015). Finding function in form: Compositional character models for open vocabulary word representation. *CoRR*, abs/1508.02096.

[Luong et al., 2013] Luong, M.-T., Socher, R., and Manning, C. D. (2013). Better word representations with recursive neural networks for morphology. In *CoNLL*, Sofia, Bulgaria.

[Mikolov et al., 2013a] Mikolov, T., Chen, K., Corrado, G., and Dean, J. (2013a). Efficient estimation of word representations in vector space. *CoRR*, abs/1301.3781.

[Mikolov et al., 2010] Mikolov, T., Karafiát, M., Burget, L., Cernocký, J., and Khudanpur, S. (2010). Recurrent neural network based language model. In *INTERSPEECH 2010, 11th Annual Conference of the International Speech Communication Association, Makuhari, Chiba, Japan, September 26-30, 2010*, pages 1045–1048.

[Mikolov et al., 2013b] Mikolov, T., Sutskever, I., Chen, K., Corrado, G., and Dean, J. (2013b). Distributed representations of words and phrases and their compositionality. *CoRR*, abs/1310.4546.

[Olah, 2015] Olah, C. (2015). Understanding lstm networks. [Online; accessed 16-May-2016].

[Sahlgren, 2008] Sahlgren, M. (2008). The distributional hypothesis. *Italian Journal of Linguistics*, 20(1):33–54.

[Schuster and Paliwal, 1997] Schuster, M. and Paliwal, K. K. (1997). Bidirectional recurrent neural networks. *IEEE TRANSACTIONS ON SIGNAL PROCESSING*, VOL. 45, NO. 11,.

[Shannon, 1948] Shannon, C. E. (1948). A Mathematical Theory of Communication. *The Bell System Technical Journal*, 27:379–423.

[Yoshua Bengio and ChristianJanvin, 2003] Yoshua Bengio, Réjean Ducharme, P. V. and ChristianJanvin (2003). A neural probabilistic language model. *J. Mach. Learn. Res.*, 3:1137–1155.