Character-Word LSTM Language Models

Lyan Verwimp **Joris Pelemans** **Hugo Van hamme** **Patrick Wambacq**
ESAT – PSI, KU Leuven
Kasteelpark Arenberg 10, 3001 Heverlee, Belgium
`firstname.lastname@esat.kuleuven.be`

Abstract

We present a Character-Word Long Short-Term Memory Language Model which both reduces the perplexity with respect to a baseline word-level language model and reduces the number of parameters of the model. Character information can reveal structural (dis)similarities between words and can even be used when a word is out-of-vocabulary, thus improving the modeling of infrequent and unknown words. By concatenating word and character embeddings, we achieve up to 2.77% relative improvement on English compared to a baseline model with a similar amount of parameters and 4.57% on Dutch. Moreover, we also outperform baseline word-level models with a larger number of parameters.

1 Introduction

Language models (LMs) play a crucial role in many speech and language processing tasks, among others speech recognition, machine translation and optical character recognition. The current state of the art are recurrent neural network (RNN) based LMs (Mikolov et al., 2010), and more specifically long short-term memory models (LSTM) (Hochreiter and Schmidhuber, 1997) LMs (Sundermeyer et al., 2012) and their variants (e.g. gated recurrent units (GRU) (Cho et al., 2014)). LSTMs and GRUs are usually very similar in performance, with GRU models often even outperforming LSTM models despite the fact that they have less parameters to train. However, Jozefowicz et al. (2015) recently showed that for the task of language modeling LSTMs work better than GRUs, therefore we focus on LSTM-based LMs.

In this work, we address some of the drawbacks of NN based LMs (and many other types of LMs). A first drawback is the fact that the parameters for infrequent words are typically less accurate because the network requires a lot of training examples to optimize the parameters. The second and most important drawback addressed is the fact that the model does not make use of the internal structure of the words, given that they are encoded as one-hot vectors. For example, ‘felicity’ (great happiness) is a relatively infrequent word (its frequency is much lower compared to the frequency of ‘happiness’ according to Google Ngram Viewer (Michel et al., 2011)) and will probably be an out-of-vocabulary (OOV) word in many applications, but since there are many nouns also ending on ‘ity’ (ability, complexity, creativity . . .), knowledge of the surface form of the word will help in determining that ‘felicity’ is a noun. Hence, subword information can play an important role in improving the representations for infrequent words and even OOV words.

In our character-word (CW) LSTM LM, we concatenate character and word embeddings and feed the resulting character-word embedding to the LSTM. Hence, we provide the LSTM with information about the structure of the word. By concatenating the embeddings, the individual characters (as opposed to e.g. a bag-of-characters approach) are preserved and the order of the characters is implicitly modeled. Moreover, since we keep the total embedding size constant, the ‘word’ embedding shrinks in size and is partly replaced by character embeddings (with a much smaller vocabulary and hence a much smaller embedding matrix), which decreases the number of parameters of the model.

We investigate the influence of the number of characters added, the size of the character embeddings, weight sharing for the characters and the size of the (hidden layer of the) model. Given that common or similar character sequences do not always occur at the beginning of words (e.g. ‘overfitting’ – ‘underfitting’), we also examine adding the characters in forward order, backward order or both orders.

We test our CW LMs on both English and Dutch. Since Dutch has a richer morphology than English due to among others its productive compounding (see e.g. (Réveil, 2012)), we expect that it should benefit more from a LM augmented with formal/morphological information.

The contributions of this paper are the following:

1. We present a method to combine word and subword information in an LSTM LM: concatenating word and character embeddings. As far as we know, this method has not been investigated before.
2. By decreasing the size of the word-level embedding (and hence the huge word embedding matrix), we effectively reduce the number of parameters in the model (see section 3.3).
3. We find that the CW model both outperforms word-level LMs with the same number of hidden units (and hence a larger number of parameters) and word-level LMs with the same number of parameters. These findings are confirmed for English and Dutch, for a small model size and a large model size. The size of the character embeddings should be proportional to the total size of the embedding (the concatenation of characters should not exceed the size of the word-level embedding), and using characters in the backward order improves the perplexity even more (see sections 3.1, 4.3 and 4.4).
4. The LM improves the modeling of OOV words by exploiting their surface form (see section 4.7).

The remainder of this paper is structured as follows: first, we discuss related work (section 2); then the CW LSTM LM is described (section 3) and tested (section 4). Finally, we give an overview of the results and an outlook to future work (section 5).

2 Related work

Other work that investigates the use of character information in RNN LMs either completely replaces the word-level representation by a character-level one or combines word and character information. Much research has also been done on modeling other types of subword information (e.g. morphemes, syllables), but in this discussion, we limit ourselves to characters as subword information.

Research on replacing the word embeddings entirely has been done for neural machine translation (NMT) by Ling et al. (2015) and Costa-jussà and Fonollosa (2016), who replace word-level embeddings with character-level embeddings. Chung et al. (2016) use a subword-level encoder and a character-level decoder for NMT. In dependency parsing, Ballesteros et al. (2015) achieve improvements by generating character-level embeddings with a bidirectional LSTM. Xie et al. (2016) work on natural language correction and also use an encoder-decoder, but operate for both the encoder and the decoder on the character level.

Character-level word representations can also be generated with convolutional neural networks (CNNs), as Zhang et al. (2015) and Kim et al. (2016) have proven for text classification and language modeling respectively. Kim et al. (2016) achieve state-of-the-art results in language modeling for several languages by combining a character-level CNN with highway (Srivastava et al., 2015) and LSTM layers. However, the major improvement is achieved by adding the highway layers: for a small model size, the purely character-level model without highway layers does not perform better than the word-level model (perplexity of 100.3 compared to 97.6), even though the character model has two hidden layers of 300 LSTM units each and is compared to a word model of two hidden layers of only 200 units (in order to keep the number of parameters similar). For a model of larger size, the character-level LM improves the word baseline (84.6 compared to 85.4), but the largest improvement is achieved by adding two highway layers (78.9). Finally, Jozefowicz et al. (2016) also describe character embeddings generated by a CNN, but they test on the 1B Word Benchmark, a data set of an entirely different scale than the one we use.

Other authors combine the word and character information (as we do in this paper) rather than doing away completely with word inputs. Chen et al. (2015) and Kang et al. (2011) work on models combining words and Chinese characters to learn embeddings. Note however that Chinese characters more closely match subwords or words than phonemes. Bojanowski et al. (2015) operate on the character level but use knowledge about the context words in two variants of character-level RNN LMs. Dos Santos and Zadrozny (2014) join word and character representations in a deep neural network for part-of-speech tagging. Finally, Miyamoto and

Cho (2016) describe a LM that is related to our model, although their character-level embedding is generated by a bidirectional LSTM and we do not use a gate to determine how much of the word and how much of the character embedding is used. However, they only compare to a simple baseline model of 2 LSTM layers of each 200 hidden units without dropout, resulting in a higher baseline perplexity (as mentioned in section 4.3, our CW model also achieves larger improvements than reported in this paper with respect to that baseline).

We can conclude that in various NLP tasks, characters have recently been introduced in several different manners. However, the models investigated in related work are either not tested on a competitive baseline (Miyamoto and Cho, 2016) or do not perform better than our models (Kim et al., 2016). In this paper, we introduce a new and straightforward manner to incorporate characters in a LM that (as far as we know) has not been investigated before.

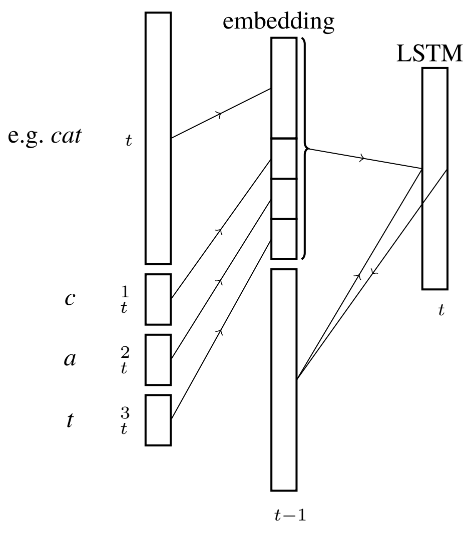

Figure 1: Concatenating word and character embeddings in an LSTM LM.

3 Character-Word LSTM LMs

A word-level LSTM LM works as follows: a word encoded as a one-hot column vector $\mathbf{w}_t$ (at time step t) is fed to the input layer and multiplied with the embedding matrix $\mathbf{W}_w$, resulting in a word embedding $\mathbf{e}_t$:

$$\mathbf{e}_t = \mathbf{W}_w \times \mathbf{w}_t \qquad (1)$$

The word embedding of the current word $\mathbf{e}_t$ will be the input for a series of non-linear operations in the LSTM layer (we refer to (Zaremba et al., 2015) for more details about the equations of the LSTM cell). In the output layer, probabilities for the next word are calculated based on a softmax function.

In our character-word LSTM LM, the only difference with the baseline LM is the computation of the 'word' embedding, which is now the result of word and character input rather than word input only. We concatenate the word embedding with embeddings of the characters occurring in that word:

$$\begin{aligned}\mathbf{e}_t^\top = [(\mathbf{W}_w \times \mathbf{w}_t)^\top (\mathbf{W}_c^1 \times \mathbf{c}_t^1)^\top \\ (\mathbf{W}_c^2 \times \mathbf{c}_t^2)^\top \ldots (\mathbf{W}_c^n \times \mathbf{c}_t^n)^\top]\end{aligned} \qquad (2)$$

where $\mathbf{c}_t^1$ is the one-hot encoding of the first character added, $\mathbf{W}_c^1$ its embedding matrix and n the total number of characters added to the model. The word $\mathbf{w}_t$ and its characters $\mathbf{c}_t^1, \mathbf{c}_t^2 \ldots \mathbf{c}_t^n$ are each projected onto their own embeddings, and the concatenation of the embeddings is the input for the LSTM layer. By concatenating the embeddings, we implicitly preserve the order of the characters: the embedding for e.g. the first character of a word will always correspond to the same portion of the input vector for the LSTM (see figure 1). We also experimented with adding word and character embeddings (a method which does not preserve the order of the characters), but that did not improve the perplexity of the LM.

The number of characters added (n) is fixed. If a word is longer than n characters, only the first (or last, depending on the order in which they are added) n characters are added. If the word is shorter than n, it is padded with a special symbol. Because we can still model the surface form of OOV words with the help of their characters, this model reduces the number of errors made immediately after OOV words (see section 4.7).

3.1 Order of the characters

The characters can be added in the order in which they appear in the word (in the experiments this is called 'forward order'), in the reversed order ('backward order') or both ('both orders'). In English and Dutch (and many other languages), suffixes can bear meaningful relations (such as plurality and verb conjugation) and compounds typically have word-final heads. Hence, putting more emphasis on the end of a word might help to better model those properties.

3.2 Weight sharing

Note that in equation 2 each position in the word is associated with different weights: the weights for the first character $\mathbf{c}_t^1$, $\mathbf{W}_c^1$, are different from the weights for the character in the second position, $\mathbf{W}_c^2$. Given that the input 'vocabulary' for characters is always the same, one could argue that the same set of weights $\mathbf{W}_c$ could be used for all positions in the word:

$$\begin{aligned}\mathbf{e}_t^\top = [&(\mathbf{W}_w \times \mathbf{w}_t)^\top (\mathbf{W}_c \times \mathbf{c}_t^1)^\top \\ &(\mathbf{W}_c \times \mathbf{c}_t^2)^\top \dots (\mathbf{W}_c \times \mathbf{c}_t^n)^\top]\end{aligned} \tag{3}$$

However, one could also argue in favor of the opposite case (no shared weights between the characters): for example, an 's' at the end of a word often has a specific meaning, such as indicating a third person singular verb form of the present tense (in English), which it does not have at other positions in the word. Both models with and without weight sharing are tested (see section 4.6).

3.3 Number of parameters

Given that a portion of the total embedding is used for modeling the characters, the actual 'word' embedding is smaller which reduces the number of parameters significantly. In a normal word-level LSTM LM, the number of parameters in the embedding matrix is

$$V \times E \tag{4}$$

with V the vocabulary size and $E = E_w$ the total embedding size/word embedding size. In our CW model however, the number of parameters is

$$V \times (E - n \times E_c) + n \times (C \times E_c) \tag{5}$$

with n the number of characters, E_c the size of the character embedding and C the size of the character vocabulary. Since V is by far the dominant factor, reducing the size of the purely word-level embedding vastly reduces the total number of parameters to train. If we share the character weights, that number becomes even smaller:

$$V \times (E - n \times E_c) + C \times E_c \tag{6}$$

4 Experiments

4.1 Setup

All LMs were trained and tested with TensorFlow (Abadi et al., 2015). We test the performance of the CW architectures for a small model and a large model, with hyperparameters based on Zaremba et al. (2015) and Kim et al. (2016)). The small LSTM consists of 2 layers of 200 hidden units and the large LSTM has 2 layers of 650 hidden units. The total size of the embedding layer always equals the size of the hidden layer. During the first 4/6 (small/large model) epochs, the learning rate is 1, after which we apply an exponential decay:

$$\eta_i = \alpha \, \eta_{i-1} \tag{7}$$

where η_i is the learning rate at epoch i and α the learning rate decay, which is set to 0.5 for the small LSTM and to 0.8 for the large LSTM. The smaller α, the faster the learning rate decreases. The total number of epochs is fixed to 13/39 (small/large model). During training, 25% of the neurons are dropped (Srivastava et al., 2014) for the small model and 50% for the large model. The weights are randomly initialized to small values (between -0.1 and 0.1 for the small model and between -0.05 and 0.05 for the large model) based on a uniform distribution. We train on mini-batches of 20 with backpropagation through time, where the network is unrolled for 20 steps for the small LSTM and 35 for the large LSTM. The norm of the gradients is clipped at 5 for both models.

For English, we test on the publicly available Penn Treebank (PTB) data set, which contains 900k word tokens for training, 70k word tokens as validation set and 80k words as test set. This data set is small but widely used in related work (among others Zaremba et al. (2015) and Kim et al. (2016)), enabling the comparison between different models. We adopt the same pre-processing as used by previous work (Mikolov et al., 2010) to facilitate comparison, which implies that the dataset contains only lowercase characters (the size of the character vocabulary is 48). Unknown words are mapped to ⟨*unk*⟩, but since we do not have the original text, we cannot use the characters of the unknown words for PTB.

The Dutch data set consists of components g, h, n and o of the Corpus of Spoken Dutch (CGN) (Oostdijk, 2000), containing recordings of meetings, debates, courses, lectures and read text. Approximately 80% was chosen as training set (1.4M word tokens), 10% as validation set (180k word tokens) and 10% as test set (190k word tokens). The size of the Dutch data set is chosen to be similar to the size of the English data set. We also use the same vocabulary size as used for Penn Treebank (10k), since we want to compare the performance on different languages and exclude any effect of the vocabulary size. However, we do not convert all uppercase characters to lowercase (although the data is normalized such that sentence-initial words with a capital are converted to lowercase if necessary) because the fact that a character is uppercase is meaningful in itself. The character vocabulary size is 88 (Dutch also includes more accented characters due to French loan words, e.g. 'café'). Hence, we do not only compare two different languages but also models with only lowercase characters and models with both upper- and lowercase characters. Moreover, since we have the original text at our disposal (as opposed to PTB), we can use the characters of the unknown words and still have a character-level representation.

4.2 Baseline models

In our experiments, we compare the CW model with two word-level baselines: one with the same number of hidden units in the LSTM layers (thus containing more parameters) and one with approximately the same number of parameters as the CW model (like Kim et al. (2016) do), because we are interested in both reducing the number of parameters and improving the performance. For the latter baseline, this implies that we change the number of hidden units from 200 to 175 for the small model and from 650 to 475 for the large, keeping the other hyperparameters the same.

The number of parameters for those models is larger than for all CW models except when only 1 or 2 characters are added. The size difference between the CW models and the smaller word-level models becomes larger if more characters are added, if the size of the characters embeddings is larger and if the character weights are shared. The size of the embedding matrix for a word-level LSTM of size 475 is 10,000 $\times$ 475 = 475,000 (V is 10k in all our experiments), whereas for a CW model with 10 character embeddings of size 25 it is of size 10,000 $\times$ (650 - 10 $\times$ 25) + 10 $\times$ (48 $\times$ 25) = 412,000 (the size of the character vocabulary for PTB is 48), following equation 5. If the character weights are shared, the size of the embedding matrix is only 401,200 (equation 6).

The baseline perplexities for the smaller word-level models are shown in table 1. In the remainder of this paper, 'wx' = means word embeddings of size x for a word-level model and 'cx' means character embeddings of size x for CW models.

			Perplexity	
Corpus	Size		Validation	Test
PTB	small	w200	100.7	96.86
		w175	102.62	98.82
	large	w650	87.38	83.6
		w465	88.39	84.38
CGN	small	w200	69.13	76
		w175	69.6	76.78
	large	w650	63.36	70.69
		w475	63.88	70.88

Table 1: Perplexities for the baseline models. Baselines w200 and w650 have the same number of hidden units as the CW models and baselines w175 and w475 approximately have the same number of parameters as the CW models.

4.3 English

In figure 2, the results for a small model trained on Penn Treebank are shown. Almost all CW models outperform the word-based baseline with the same number of parameters (2 LSTM layers of 175 units). Only the CW models in which the concatenated character embeddings take up the majority of the total embedding (more than 7 characters of embedding size 15) perform worse. With respect to the word-level LM with more parameters, only small improvements are obtained. The smaller the character embeddings, the better the performance of the CW model. For example, for a total embedding size of only 200, adding 8 character embeddings of size 15 results in an embedding consisting of 120 units 'character embedding' and only 80 units 'word embedding', which is not sufficient. The two best performing models add 3 and 7 character embeddings of size 5, giving a perplexity of 100.12 and 100.25 respectively, achieving a relative improvement of 2.44%/2.31% w.r.t. the w175 baseline and 0.58%/0.45% w.r.t. the w200 baseline. For those models, the 'word embedding' consists of 185 and 165 units respectively.

We test the performance of the CW architecture on a large model too. In figure 3, the results for different embedding sizes are shown. Just like we saw for the small model, the size of the character embeddings should not be too large: for embeddings of size 50 ('c50'), the performance drops when a larger number of characters is added. The best result is obtained by adding 8 characters with embeddings of size 25 ('c25'): a perplexity of 85.97 (2.74%/1.61% relative improvement with respect to

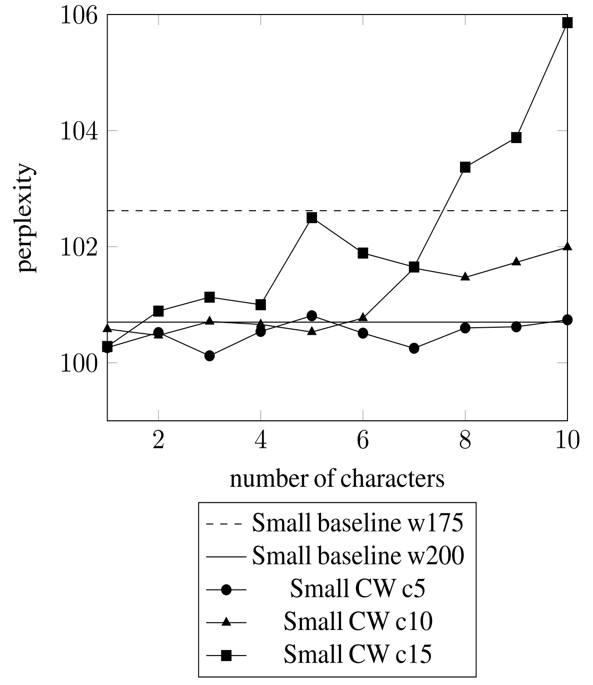

Figure 2: Validation perplexity results on PTB, small model. Different sizes for the character embeddings are tested ('c5', 'c10', 'c15').

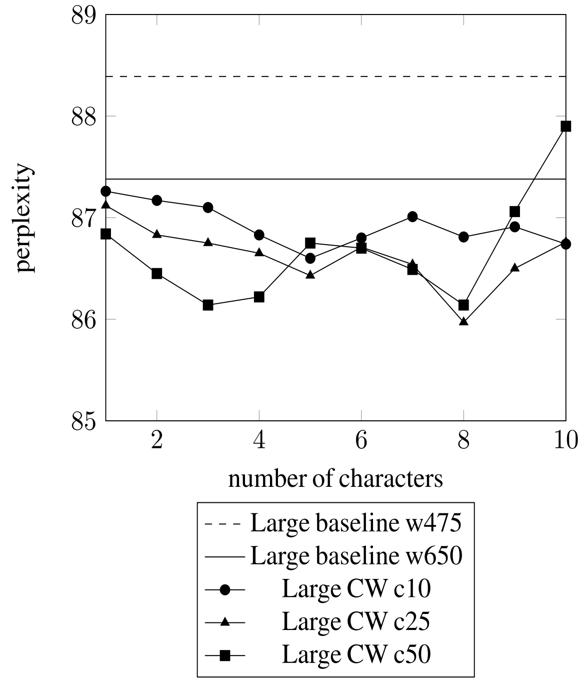

Figure 3: Validation perplexity results on PTB, large model. Different sizes for the character embeddings are tested ('c10', 'c25', 'c50').

the w475/w650 baseline). For embeddings of size 10, adding more than 10 characters gives additional improvements (see figure 4).

We also verify whether the order in which the characters are added is important (figure 4). The best result is achieved by adding the first 3 and the last 3 characters to the model ('both orders'), giving a perplexity of 85.69, 3.05%/1.87% relative improvement with respect to the w475/w650 baseline. However, adding more characters in both orders causes a decrease in performance. When only adding the characters in the forward order or the backward order, adding the characters in backward order seems to perform slightly better overall (best result: adding 9 characters in the backward order gives a perplexity of 85.7 or 3.04%/1.92% improvement with respect to the w475/w650 baseline).

We can conclude that the size of the character embeddings should be proportional to the total embedding size: the word-level embedding should be larger than the concatenation of the character-level embeddings. Adding characters in the backward order is slightly better than adding them in the forward order, and the largest improvement is made for the large LSTM LM. The test perplexities for some of the best performing models (table 2) confirm these findings.

If we compare the test perplexities with two related models that incorporate characters, we see that our models perform better. Kim et al. (2016) generate character-level embeddings with a convolutional neural network and also report results for both a small and a large model. Their small character-level model has more hidden units than ours (300 compared to 200), but it does not improve with respect to the word-level baseline (since we do not use highway layers, we only compare with the results for models without highway layers). Their large model slightly improves their own baseline perplexity (85.4) by 0.94%. Compare with our results: 2.64% perplexity reduction for the best small LSTM (c5 with $n=3$) and 2.77% for the best large LSTM (c10 with $n = 3 + 3(b)$). Miyamoto and Cho (2016) only report results for a small model that is trained without dropout, resulting in a baseline perplexity of 115.65. If we train our small model without dropout we get a comparable baseline perplexity (116.33) and a character-word perplexity of 110.54 (compare to 109.05 reported by Miyamoto and Cho (2016)). It remains to be seen whether their model performs equally well compared to better baselines. Moreover, their hybrid character-word model is more complex than ours because it uses a bidirectional LSTM to generate

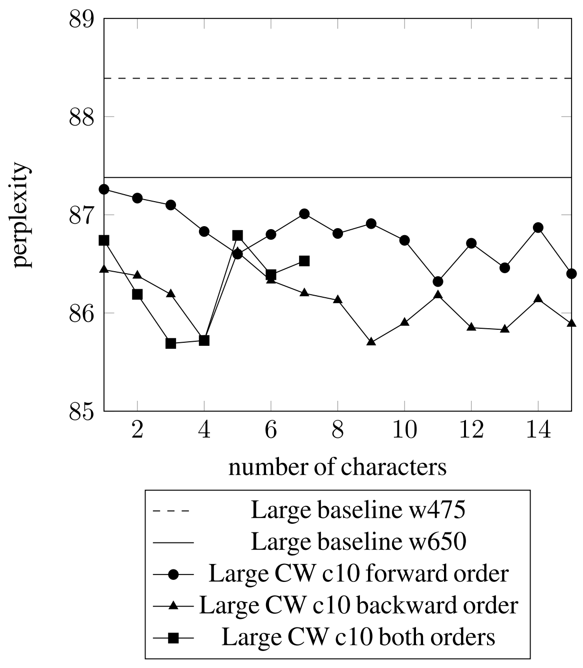

Figure 4: Validation perplexity results on PTB, large model. Several options for the order in which the characters are added are investigated.

Small model	**Perplexity**
Baseline w175/w200	98.82/96.86
(Kim et al., 2016)	100.3
(Miyamoto and Cho, 2016)	109.05
c5 with $n = 3$	**96.21**
c5 with $n = 7$	96.35
Large model	**Perplexity**
Baseline w475/w650	84.38/83.6
(Kim et al., 2016)	84.6
c25 with $n = 8$	82.69
c10 with $n = 9(b)$	82.68
c10 with $n = 3 + 3(b)$	**82.04**

Table 2: Test perplexity results for the best models on PTB. Baseline perplexities are for sizes w175/w200 for a small model and w475/w650 for a large model. n = number of characters added, (b) means backward order. Comparison with other character-level LMs (Kim et al., 2016) (we only compare to models without highway layers) and character-word models (Miyamoto and Cho, 2016) (they do not use dropout and only report results for a small model).

the character-level embedding (instead of a lookup table) and a gate to determine the mixing weights between the character- and word-level embeddings.

4.4 Dutch

As we explained in the introduction, we expect that using information about the internal structure of the word will help more for languages with a richer morphology. Although Dutch is still an analytic language (most grammatical relations are marked with separate words or word order rather than morphemes), it has a richer morphology than English because compounding is a productive and widely used process and because it has more lexical variation due to inflection (e.g. verb conjugation, adjective inflection). The results for the LSTM LMs trained on Dutch seem to confirm this hypothesis (see figure 5).

The CW model outperforms the baseline word-level LM both for the small model and the large model. The best result for the small model is obtained by adding 2 or 3 characters, giving a perplexity of 67.59 or 67.65 which equals a relative improvement of 2.89%/2.23% (w175/w200) and 2.80%/2.14% (w175/w200) respectively.

For the large model, we test several embedding sizes and orders for the characters. The best model is the one to which 6 characters in backward order are added, with a perplexity of 60.88 or 4.70%/3.91% (w475/w650) relative improvement. Just like for PTB, an embedding size of 25 proves to be the best compromise: if the characters are added in the normal order, 4 characters with embeddings of size 25 is the best model (perplexity 61.47 or 3.77%/2.98% (w475/w650) relative improvement).

These results are confirmed for the test set (table 3). The best small model has a perplexity of 75.04 which is 2.27% compared to the baseline and the best large model has a perplexity of 67.64, a relative improvement of 4.57%. The larger improvement for Dutch might be due to the fact that it has a richer morphology and/or the fact that we can use the surface form of the OOV words for the Dutch data set (see sections 4.1 and 4.7).

4.5 Random CW models

In order to investigate whether the improvements of the CW models are not caused by the fact that the characters add some sort of noise to the input, we experiment with adding real noise – random 'character' information – rather than the real characters. Both the number of characters (the length of the random 'word') and the 'characters' themselves are generated based on a uniform distribution. In table 4, the relative change in perplexity, averaged over models to which 1 to 10 characters are added,

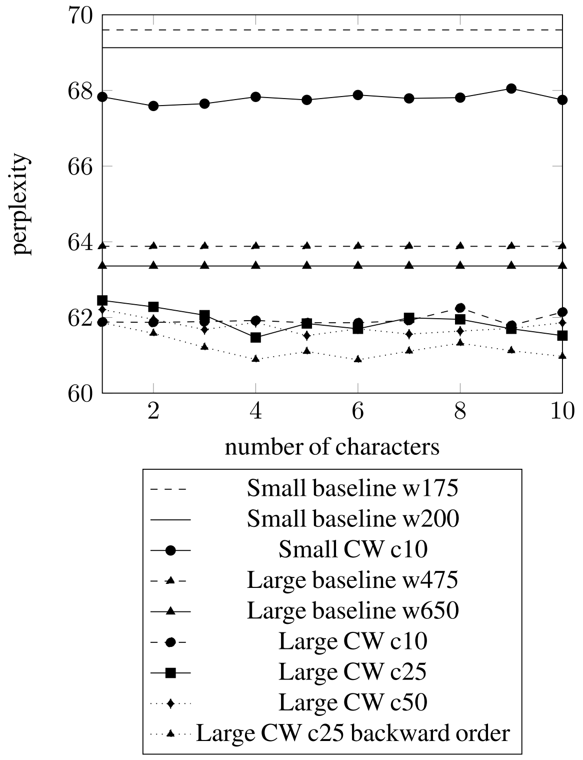

Figure 5: Validation perplexity results on CGN. Several options for the size and order of the character embeddings are investigated.

Small model	**Perplexity**
Baseline w175/w200	76.78/76
c10 with $n=2$	75.23
c10 with $n=3$	**75.04**
Large model	**Perplexity**
Baseline w475/w650	70.88/70.69
c25 with $n=4$	68.79
c25 with $n=6(b)$	**67.64**

Table 3: Test perplexity results for the best models on CGN. Baseline perplexities are for sizes w175/w200 for a small model and w475/w650 for a large model. n = number of characters added, (b) means backward order.

with respect to the baseline word-level LM and the CW model with real characters is shown.

For English, adding random information had a negative impact on the performance with respect to both the baseline and the CW model. For Dutch on the other hand, adding some random noise to the word-level model gave small improvements. However, the random models perform much worse than the CW models. We can conclude that the characters provide meaningful information to the LM.

		Relative change in valid perplexity w.r.t. **Baseline**	**Char-Word**
PTB	small c5	0.34 (0.30)	0.54 (0.46)
	large c15	0.00 (0.29)	0.53 (0.49)
CGN	small c10	- 0.18 (0.53)	1.79 (0.47)
	large c10	- 0.15 (0.26)	1.52 (1.24)

Table 4: Relative change in validation perplexity for models to which **random** information is added, w.r.t. word-level and CW models. The improvements are averaged over the results for adding 1 to 10 characters/random information, the numbers between brackets are standard deviations. Negative numbers indicate a decrease in perplexity.

4.6 Sharing weights

We repeat certain experiments with the CW models, but with embedding matrices that are shared across all character positions (see section 3.2). Note that sharing the weights does not imply that the position information is lost, because for example the first portion of the character-level embedding always corresponds to the character on the first position. Sharing the weights ensures that a character is always mapped onto the same embedding, regardless of the position of that character in the word, e.g. both occurrences of ‘i’ in ‘felicity’ are represented by the same embedding. This effectively reduces the number of parameters.

We compare the performance of the CW models with weight sharing with the baseline word-level LM and the CW model without weight sharing. In table 5, the relative change with respect to those LMs is listed.

CW models with weight sharing generally improve with respect to a word-level baseline, except for the small English LM. For Dutch, the improvements are more pronounced. The difference with the CW model without weight sharing is small (right column), although *not* sharing the weights works slightly better, which suggests that characters can convey different meanings depending on the position in which they occur. Again, the results are more clear-cut for Dutch than for English.

4.7 Dealing with out-of-vocabulary words

As we mentioned in the introduction, we expect that by providing information about the surface form of OOV words (namely, their characters), the number of errors induced by those words should decrease.

		Relative change in valid perplexity w.r.t.	
		Baseline	**Char-Word**
PTB	small c10	0.53 (0.88)	0.19 (0.67)
	large c10	- 0.54 (0.37)	- 0.02 (0.22)
CGN	small c10	- 1.70 (0.34)	0.24 (0.30)
	large c10	- 2.10 (0.32)	0.15 (0.50)

Table 5: Relative change in validation perplexity for CW models with **weight sharing** for the characters, w.r.t. baseline and CW models without weight sharing. The improvements are averaged over the results for adding 1 until 10 characters, the numbers between brackets are standard deviations. Negative numbers indicate a decrease in perplexity.

We conduct the following experiment to check whether this is indeed the case: for the CGN test set, we keep track of the probabilities of each word during testing. If an OOV word is encountered, we check the probability of the target word given by a word-level LM and a CW LM. The word-level model is a large model of size 475 and the CW model is a large model in which 6 characters embeddings of size 25 in the backward order are used (the best performing CW model in our experiments).

We observe that in 17,483 of the cases, the CW model assigns a higher probability to the target word following an OOV word, whereas the opposite happens only in 10,724 cases. This is an indication that using the character information indeed helps in better modeling the OOV words.

5 Conclusion and future work

We investigated a character-word LSTM language model, which combines character and word information by concatenating the respective embeddings. This both reduces the size of the LSTM and improves the perplexity with respect to a baseline word-level LM. The model was tested on English and Dutch, for different model sizes, several embedding sizes for the characters, different orders in which the characters are added and for weight sharing of the characters. We can conclude that for almost all setups, the CW LM outperforms the word-level model, whereas it has fewer parameters than the word-level model with the same number of LSTM units. If we compare with a word-level LM with approximately the same number of parameters, the improvement is larger.

One might argue that using a CNN or an RNN to generate character-level embeddings is a more general approach to incorporate characters in a LM, but this model is simple, easier to train and smaller. Moreover, related models using a CNN-based character embedding (Kim et al., 2016) do not perform better.

For both English and Dutch, we see that the size of the character embedding is important and should be proportional to the total embedding size: the total size of the concatenated character embeddings should not be larger than the word-level embedding. Not using the characters in the order in which they appear in the word, but in the reversed order (and hence putting more emphasis on the end of the word), performs slightly better, although adding only a few characters both from the beginning and the end of the word achieves good performance too.

Using random inputs instead of the characters performed worse than using the characters themselves, thus refuting the hypothesis that the characters simply introduce noise. Sharing the weights/embedding matrices for the characters reduces the size of the model even more, but causes a small increase in perplexity with respect to a model without weight sharing. Finally, we observe that the CW models are better able to deal with OOV words than word-level LMs.

In future work, we will test other architectures to incorporate character information in a word-level LSTM LM, such as combining a character-level LSTM with a word-level LSTM. Another representation that might be useful uses character co-occurrence vectors (by analogy with the acoustic co-occurrences used by Van hamme (2008; 2012)) rather than one-hot character vectors, because co-occurrences intrinsically give information about the order of the characters. Other models could be more inspired by human language processing: according to the theory of *blocking*, we humans have both a mental lexicon of frequent words and a morphological module that is used to process infrequent/ unknown words or to create new words (see e.g. (Aronoff and Anshen, 2001)). This could correspond to a word-level LM for frequent words and a subword-level LM for infrequent words.

Acknowledgments

This work was funded by IWT-INNOVATIEF AANBESTEDEN and VRT in the STON project.

References

Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S. Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Ian Goodfellow, Geoffrey Irving Andrew Harp, Michael Isard, Rafal Jozefowicz, Yangqing Jia, Lukasz Kaiser, Manjunath Kudlur, Josh Levenberg, Dan Man, Mike Schuster, Rajat Monga, Sherry Moore, Derek Murray, Chris Olah, Jonathon Shlens, Benoit Steiner, Ilya Sutskever, Kunal Talwar, Paul Tucker, Vincent Vanhoucke, Vijay Vasudevan, Fernanda Vigas, Oriol Vinyals, Pete Warden, Martin Wattenberg, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. 2015. Tensorflow: Large-scale machine learning on heterogeneous systems. *Software available from tensorflow.org*.

Marc Aronoff and Frank Anshen. 2001. Morphology and the lexicon: Lexicalization and productivity. In Andrew Spencer and Arnold M. Zwicky, editors, *The Handbook of Morphology*, pages 237–247. Blackwell Publishing.

Miguel Ballesteros, Chris Dyer, and Noah A. Smith. 2015. Improved Transition-Based Parsing by Modeling characters instead of words with LSTMs. *Conference on Empirical Methods in Natural Language Processing (EMNLP)*, pages 349–359.

Piotr Bojanowski, Armand Joulin, and Tomáš Mikolov. 2015. Alternative structures for character-level RNNs. *arXiv:1511.06303*.

Xinxiong Chen, Lei Xu, Zhiyuan Liu, Maosong Sun, and Huanbo Luan. 2015. Joint Learning of Character and Word Embeddings. *Conference on Artificial Intelligence (AAAI)*, pages 1236–1242.

Kyunghyun Cho, Bart van Merriënboer, Caglar Gulcehre, Dzmitry Bahdanau, Fethi Bougares, Holger Schwenk, and Yoshua Bengio. 2014. Learning Phrase Representations using RNN Encoder–Decoder for Statistical Machine Translation. *Conference on Empirical Methods in Natural Language Processing (EMNLP)*, pages 1724–1734.

Junyoung Chung, Kyunghyun Cho, and Yoshua Bengio. 2016. A Character-Level Decoder without Explicit Segmentation for Neural Machine Translation. *arXiv:1603.06147*.

Marta R. Costa-jussà and José A.R. Fonollosa. 2016. Character-based Neural Machine Translation. *Proceedings of the Association for Computational Linguistics (ACL)*, 2:357–361.

Cícero N. dos Santos and Bianca Zadrozny. 2014. Learning Character-level Representations for Part-of-Speech Tagging. *Proceedings of The 31st International Conference on Machine Learning (ICML)*, pages 1818–1826.

Hugo Van hamme. 2008. HAC-models: a novel approach to continuous speech recognition. *Proceedings Interspeech*, pages 2554–2557.

Hugo Van hamme. 2012. An on-line NMF model for temporal pattern learning: Theory with application to automatic speech recognition. *International Conference on Latent Variable Analysis and Signal Separation (LVA/ICA)*, pages 306–313.

Sepp Hochreiter and Jürgen Schmidhuber. 1997. Long short-term memory. *Neural Computation*, 9(8):1735–1780.

Rafal Jozefowicz, Wojciech Zaremba, and Ilya Sutskever. 2015. An Empirical Exploration of Recurrent Network Architectures. *Proceedings of the 32nd International Conference on Machine Learning (ICML)*, pages 2342–2350.

Rafal Jozefowicz, Oriol Vinyals, Mike Schuster, Noam Shazeer, and Yonghui Wu. 2016. Exploring the Limits of Language Modeling. *arXiv:1602.02410*.

Moonyoung Kang, Tim Ng, and Long Nguyen. 2011. Mandarin word-character hybrid-input Neural Network Language Model. *Proceedings Interspeech*, pages 1261–1264.

Yoon Kim, Yacine Jernite, David Sontag, and Alexander M. Rush. 2016. Character-Aware Neural Language Models. *Proc. Conference on Artificial Intelligence (AAAI)*, pages 2741–2749.

Wang Ling, Isabel Trancoso, Chris Dyer, and Alan Black. 2015. Character-based Neural Machine Translation. *arXiv:1511.04586*.

Jean-Baptiste Michel, Yuan Kui Shen, Aviva Presser Aiden, Adrian Veres, Matthew K. Gray, William Brockman, The Google Books Team, Joseph P. Pickett, Dale Hoiberg, Dan Clancy, Peter Norvig, Jon Orwant, Steven Pinker, Martin A. Nowak, and Erez Lieberman Aiden. 2011. Quantitative Analysis of Culture Using Millions of Digitized Books. *Science*, 331(6014):176–182.

Tomáš Mikolov, Martin Karafiát, Lukáš Burget, Jan Černocký, and Sanjeev Khudanpur. 2010. Recurrent neural network based language model. *Proceedings Interspeech*, pages 1045–1048.

Yasumasa Miyamoto and Kyunghyun Cho. 2016. Gated Word-Character Recurrent Language Model. *arXiv:1606.01700*.

Nelleke Oostdijk. 2000. The Spoken Dutch Corpus. Overview and first Evaluation. *Proceedings Language Resources and Evaluation Conference (LREC)*, pages 887–894.

Bert Réveil. 2012. Optimizing the Recognition Lexicon for Automatic Speech Recognition. *PhD thesis, University of Ghent, Belgium.*

Nitish Srivastava, Geoffrey Hinton, Alex Krizhevsky, Ilya Sutskever, and Ruslan Salakhutdinov. 2014. Dropout: A Simple Way to Prevent Neural Networks from Overfitting. *Journal of Machine Learning Research*, 15:1929–1958.

Rupesh K. Srivastava, Klaus Greff, and Jürgen Schmidhuber. 2015. Training Very Deep Networks. *Neural Information Processing Systems Conference (NIPS)*, pages 2377–2385.

Martin Sundermeyer, Ralf Schlüter, and Hermann Ney. 2012. LSTM Neural Networks for Language Modeling. *Proceedings Interspeech*, pages 1724–1734.

Ziang Xie, Anand Avati, Naveen Arivzhagan, Dan Jurafsky, and Andrew Y. Ng. 2016. Neural Language Correction with Character-Based Attention. *arXiv:1603.09727*.

Wojciech Zaremba, Ilya Sutskever, and Oriol Vinyals. 2015. Recurrent Neural Network Regularization. *arXiv:1409.2329*.

Xiang Zhang, Junbo Zhao, and Yann LeCunn. 2015. Character-level Convolutional Networks for Text Classification. *Neural Information Processing Systems Conference (NIPS)*, pages 649–657.