TF-LM: TensorFlow-based Language Modeling Toolkit

Lyan Verwimp, Hugo Van hamme, Patrick Wambacq
ESAT – PSI, KU Leuven
Kasteelpark Arenberg 10, 3001 Heverlee, Belgium
{lyan.verwimp, hugo.vanhamme, patrick.wambacq}@esat.kuleuven.be

Abstract

Recently, an abundance of deep learning toolkits has been made freely available. These toolkits typically offer the building blocks and sometimes simple example scripts, but designing and training a model still takes a considerable amount of time and knowledge. We present language modeling scripts based on TensorFlow that allow one to train and test competitive models directly, by using a pre-defined configuration or changing it to their needs. There are several options for input features (words, characters, words combined with characters, character n-grams) and for batching (sentence- or discourse-level). The models can be used to test the perplexity, predict the next word(s), re-score hypotheses or generate debugging files for interpolation with n-gram models. Additionally, we make available LSTM language models trained on a variety of Dutch texts and English benchmarks, that can be used immediately, thereby avoiding the time and computationally expensive training process. The toolkit is open source and can be found at `https://github.com/lverwimp/tf-lm`.

Keywords: language modeling, LSTM, deep learning, toolkit

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 (LSTM) (Hochreiter and Schmidhuber, 1997) LMs (Sundermeyer et al., 2012). Building, training, optimizing and testing these networks from scratch would require a huge amount of expertise and time. There exist many deep learning frameworks that offer the building blocks: TensorFlow (Abadi et al., 2015), Keras (Chollet, 2015), Torch (Collobert et al., 2011), Theano (Theano Development Team, 2016), Caffe (Jia et al., 2014) and others. Researchers proposing a new type of model also frequently publish their code, but typically do not offer a more general framework.

Among the deep learning frameworks, TensorFlow is arguably the most widely used, as the Github statistics in table 1. demonstrate. However, as far as we know (see section 2. for a more detailed description of the existing tools), there does not exist a toolkit that allows one to quickly design, train and test their own 'baseline' LMs with TensorFlow. We release a toolkit that contains modular code that should be easy to adapt, standard recipes to train competitive baseline models that can be adapted in a simple configuration file and LMs that are pre-trained on a variety of English benchmarks and corpora of spoken Dutch.

Framework	Stars	Forks
TensorFlow	86,940	33,924
Caffe	20,003	12,275
Keras	19,261	6,978
Torch	7,232	2,140
Theano	6,862	2,277

Table 1: Github statistics for several deep learning frameworks, recorded on Sept 6, 2017.

In the remainder of this paper, we discuss other toolkits that provide language modeling scripts in section 2., describe the documentation and functionality that our toolkit provides (sections 3. and 4.) and the pre-trained LMs (section 5.). We end with experimental results (section 6.) and a conclusion (section 7.).

2. Related work

Although there already exist open-source neural language modeling toolkits, they only offer feedforward NNs (Schwenk, 2013), vanilla RNNs (Mikolov et al., 2014) or are not easy to adapt and hence not very attractive to researchers (Sundermeyer et al., 2014). TheanoLM (Enarvi and Kurimo, 2016) should be more flexible and offers many state-of-the-art models such as LSTMs and GRUs, but is built on Theano. TensorFlow has a larger community than Theano and is updated frequently, also including state-of-the-art models. Moreover, it has been announced that the development of Theano will not be continued.

The TensorFlow documentation offers a tutorial on recurrent neural network language modeling (TensorFlow, 2017), with code to train a word-level LM. It trains on batches that go across sentence boundaries, while not resetting the state of the LSTM for every new sentence. This implies that the model in theory can remember words from the previous sentence(s). However, for some architectures (e.g. bidirectional models) or applications, working on the sentence level is a more natural choice. As explained in section 4., we offer many more options in our toolkit, among others the choice between training on sentence-level or 'discourse'-level batches.

Several Github projects provide code to train LMs with TensorFlow, but they are often quite specialized. For example, *word-rnn-tensorflow* (Kim, 2017) and *char-rnn-tensorflow* (Ozair, 2017) allow one to train respectively word-level and character-level RNN LMs that one can use to sample text (this corresponds to what we call 'predicting the next word(s)', see section 4.4.).

To the best of our knowledge, we are not aware of a similar TensorFlow-based language modeling toolkit that offers the same variety of options as ours.

3. Documentation

Documentation about the toolkit can be found in the Github repository: we provide a high-level overview of the options the toolkit offers, an overview of what every script does, a list of possible combinations of options together with examples of commands and a link to the pre-trained models. We also give a detailed description of what the configuration should look like: mandatory versus optional parameters, expected type for every parameter and a description of what they do. The web page containing the pre-trained LSTM LMs also gives some information about the data and models.

4. Functionality

TF-LM offers different options with respect to device placement, data reading, unit of input/output, batching, training schemes and testing.

Firstly, the code will automatically run on GPU if one is available, but if one does not want to use the available GPU, it is possible to run on CPU by simply using the command line option *--device cpu*.

Secondly, with respect to data handling, all data is read at once and kept in memory, but this is not possible when dealing with large datasets. For those datasets one can choose to read the data sentence per sentence. This option can so far only be combined with sentence-level batches (see section 4.2. for more information about batching).

Thirdly, the main script will automatically train, validate and test the model specified in the configuration file, unless certain options are switched off (with command line options *--train False*, *--valid False* or *--test False*). Thus, if one for example only wants to check the perplexity of a certain data set for an already trained model, this can be done by using these switches.

All other options that the toolkit offers are specified in a configuration file (command line option *--config*), which contains a list of parameters and values. Certain parameters should always be specified, such as the path where the model should be saved and the path where the data can be found, others are optional. More detailed information about every option, and about the possible combinations of options can be found in the Github READMEs.

We tried to design the toolkit in such a manner that implementing new models or new manners of feeding data, training or testing the models should be easy. One can typically start from a base class and adapt only the parts that are necessary: e.g. the LM that takes as input character n-grams rather than words inherits from the baseline LM class, and only adapts the initialization of variables and the manner in which the input embeddings are generated.

4.1. Words, characters or both?

As regards unit of input/output, there are four options: word, character, word and characters and character n-grams. The word-level LM is the default option, and trains a model that predicts the next word given the previous words. The input embedding $\mathbf{e}_t$ for the word-level model is calculated as follows:

$$\mathbf{e}_t = \mathbf{W}_w \times \mathbf{w}_t \tag{1}$$

with $\mathbf{W}_w$ the word embedding matrix and $\mathbf{w}_t$ the one-hot vector of the word at time step t. The character-level LM does exactly the same but for characters: $\mathbf{c}_t$ is the one-hot vector of the current character.

$$\mathbf{e}_t = \mathbf{W}_c \times \mathbf{c}_t \tag{2}$$

In the output layer, this model predicts the next character.

The combination of word and characters feeds the current word to the input along with a predetermined number of characters occurring in the word, as in (Verwimp et al., 2017b). The word and character embeddings are concatenated and the result of this operation is fed to the LSTM:

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

where $\mathbf{c}_t^1$ is the one-hot encoding of the first character, $\mathbf{W}_c^1$ its embedding matrix and n the total number of characters added to the model. This option can be specified with the parameter *word_char_concat* (set to *True*), the number of characters to be added with *num_char*, the size of the character embeddings with *char_size* and the order in which the characters should be added with *order*: *begin_first* implies that we start at the beginning of the word (e.g. 5 characters from 'pineapple': *p, i, n, e, a*), *end_first* that we start at the ending of the word (e.g. 5 characters from 'pineapple': *e, l, p, p, a*), and *both* that we add both the first *num_char* characters starting from the beginning and from the ending (e.g. 3 characters from 'pineapple': *p, i, n; e, l, p*). In the output layer the model predicts the next word.

The option of character n-grams (parameter *char_ngram*) feeds a vector containing the counts of all character n-grams that occur in the current word to the network:

$$\mathbf{e}_t = \mathbf{W}_g \times \mathbf{g}_t \tag{4}$$

where $\mathbf{g}_t$ is *not* a one-hot vector but a vector of the length of the character n-gram vocabulary, containing for every character n-gram its frequency in the current word. For example, for the word 'home', the character 2-grams are: *<bow>h, ho, om, me and e<eow>* (we append a beginning- and end-of-word token *<bow>, <eow>*). It is also possible to use skipgrams (parameter *skipgram* with an integer value specifying the number of skips), for example with skips of 1 character, 'home' has the following skipgrams: *<bow>o, hm, oe, m<eow>*.

In this model, the representation needs more memory, since every word is represented by a vector instead of an index, which is the implicit representation of a one-hot vector. To restrict the size of the input vocabulary, one can choose to set a frequency cutoff for the character n-grams (*ngram_cutoff*): all n-grams not in the resulting vocabulary are mapped to an *unknown-ngram*-symbol. Another option to reduce the vocabulary is to only use lowercase characters

and add a special symbol <*cap*> to the vector and assign it the number of capitals in the word (parameter *capital*).
Finally, it is also possible to reserve a part of the total input embedding for a standard word embedding, in a similar manner as for character-word LMs (Verwimp et al., 2017b):

$$\mathbf{e}_t^\top = [(\mathbf{W}_w \times \mathbf{w}_t)^\top (\mathbf{W}_g \times \mathbf{g}_t)^\top] \quad (5)$$

In this case, the input embedding consists of a concatenation of the word embedding and the character n-gram input. In the output layer, the character n-gram model always predicts a distribution over words.

4.2. Going beyond sentence boundaries?

By default, the models are trained on batches that optimally make use of the available space. For example (extract taken from Penn TreeBank):

> *"owned by <unk> & <unk> co. was under contract with <unk> to make the cigarette filters <eos> the finding probably"*

The batches may contain (multiple) (parts of a) sentence, separated from each other with the end-of-sentence token <*eos*>. Since the hidden state of the LSTM is transferred across the batches, it is not a problem that sentences are spread over several batches and the LSTM can in theory remember information from previous sentences. If the parameter *per_sentence* is present in the configuration file, the model will train on batches that contain only one sentence, padded until they all have the same length:

> *"<bos> the plant which is owned by <unk> & <unk> co. was under contract with <unk> to make the cigarette filters <eos> @ @ @ ..."*
>
> *"<bos> the finding probably will support those who argue that the u.s. should regulate the class of asbestos including <unk> more <unk> than the common kind of asbestos <unk> found in most schools and other buildings dr. <unk> said <eos> @ @ @ ..."*

In the above examples, '@' is the padding symbol and the number of padding symbols that has to be added to obtain the length of the longest sentence in the whole dataset is often large. We also introduce a beginning-of-sentence symbol <*bos*> to be able to predict the first word in the sentence. In this model, a lot of memory is wasted on padding, although in principle no extra computation is done using TensorFlow's dynamic RNN. To reduce the memory usage, it is possible to stream the data sentence per sentence. The state of the LSTM is always reset after each sentence, such that it does not remember the previous sentence(s). The perplexity calculation is adapted to exclude the padding. We will refer to this condition as 'sentence LSTM', whereas the one that should remember previous sentences is referred to as 'discourse LSTM'.
It is important to note that most research papers on language modeling do not explicitly mention whether their models go beyond sentence boundaries or not. Some exceptions are (Pelemans et al., 2016) and (Chelba et al., 2017), who report worse perplexity results for models trained on sentence level than on discourse level, supporting the intuition that knowledge of the previous sentence(s) can have a positive influence on the language modeling capacity. However, for certain architectures or applications, such as bidirectional models, sentence-level models are required.

4.3. Training

Certain parameters for training always have to be specified in the configuration file. Firstly, the vocabulary size should be specified (*vocab_size*) and if it is smaller than the full vocabulary of the data, words that do not appear in the vocabulary should be mapped to an *unknown* token (we also make available a script to do this mapping). All models are trained with an open vocabulary: the *unknown* token is part of the input and output vocabulary and hence an 'unknown' word can be predicted. If one does not want to use a vocabulary based on the frequency of the words, one can load their own vocabulary with the option *read_vocab_from_file*.
Secondly, the size of the TensorFlow graph can be specified in number of layers, number of neurons per layer, batch size and number of steps to unroll the network for backpropagation through time. The models are randomly initialized with a uniform distribution between *-init_scale* and *+init_scale*. Several optimizers are included (stochastic gradient descent, adam and adagrad) but a new optimizer can be easily added. With respect to regularization, by default dropout is used on the input embeddings and on the outputs of the LSTM cell, and the norm of the gradients is clipped to avoid exploding gradients.
If the configuration parameter *bidirectional* is added, the LM is trained as a bidirectional LSTM, whereby the forward state of the current time step and the backward state of next time step + 2 are combined:

$$y_{t+1} = f(W_o^f \; h_t^f + W_o^b \; h_{t+2}^b + b) \quad (6)$$

In the equation above, f is the softmax function, y_{t+1} is the probability distribution predicted for the word at time step $t+1$, W_o^f and W_o^b the output weight matrices of respectively the forward and backward LSTM, h_t^f the hidden state of the forward LSTM for the current time step, h_{t+2}^b the hidden state of the backward LSTM for the next time step + 2 and b the output bias vector. We use h_{t+2}^b because for the task of language modeling, the input from the next time step is equal to the target from the current time step.
There are two training schemes available: training a fixed number of epochs or training with early stopping. For early stopping, the validation perplexity of the current epoch is compared with those of the x (specified with *early_stop*) previous epochs, and training is stopped when it has been higher x times. If this is not the case, training is continued until the maximum number of epochs (*max_max_epoch*), similar to the other training scheme. Both schemes can be combined with an exponentially decaying learning rate: during the first epochs (specified with *max_epoch*), the same initial learning rate is used. Then an exponential decay is applied:

$$\eta_i = \alpha \; \eta_{i-1} \quad (7)$$

where η_i is the learning rate at epoch i and α the learning rate decay.

4.4. Testing

TF-LM offers several options to test the performance of a trained LM. Firstly, the test or validation perplexity of the standard data sets can be calculated by running the same configuration with the *--train* and respectively *--valid* or *--test* arguments switched off. Other test sets can be specified with the *other_test* parameter.

Secondly, a trained model can be used for re-scoring sentences: given a list of sentences/hypotheses, the model will assign log probabilities to every sentence, that can be used for re-ranking hypotheses.

A third option is generating the most likely (sequence of) word(s) given a certain history according to a trained model. This option can be specified with *predict_next* and takes as input a file containing word sequences that serve as the 'history'. It then iteratively samples the most likely word given the history, then given the history including the previously sampled word, and so on. It is possible to specify the maximum number of words that should be generated with *max_num_predictions* (the default is 100). The generation will stop if the <*eos*>-symbol is predicted or else if the maximum number of predictions has been generated. Optionally, one can choose to not generate the most likely word at every step, but to sample from a multinomial distribution as specified by the softmax probabilities.

The final option for testing can be used to easily calculate interpolation weights for LSTM LMs and n-gram LMs. SRILM (Stolcke, 2002) offers a script to calculate the optimal interpolation weights between several models (*compute-best-mix*) based on the outputs of running different LMs on the same test set (with the *-debug 2* option, which prints the probabilities per word). The debug file generated by our code has a similar structure as SRILM's debug file, containing (log) probabilities per word.

Re-scoring, generating the next word(s) and generating a debug file are all implemented on sentence-level batches, even if the LM was trained on discourse-level batches.

5. Pre-trained LSTM Language Models

We make available LSTM LMs trained on several English and Dutch datasets, that can be found at `http://homes.esat.kuleuven.be/~lverwimp/lstm_lm/`.

The English datasets are two publicly available benchmarks: Penn TreeBank (PTB) (Marcus et al., 1993) and WikiText (Wiki) (Merity et al., 2016). PTB contains 900k word tokens for training, 70k word tokens as validation set and 80k words as test set and has a vocabulary of 10k words. For WikiText we used the small dataset for training, WikiText-2, which contains 2M words for training, 210k words for validation and 240k words for testing. We used the same 33k vocabulary as Merity et al. (2016).

The Dutch datasets are two corpora of spoken Dutch, the Corpus of Spoken Dutch (CGN) (Oostdijk, 2000) and a dataset of subtitles (Sub). The CGN dataset contains normalized versions of all components, both Flemish and Dutch. The dataset is split in a training set of 8M words, a validation set of 200k words and a test set of 240k words. The Sub dataset contains 45M words for training, 100k words for validation and 120k words for testing. It contains subtitles for a variety of TV shows from the Flemish national broadcaster, including fiction, documentaries, talk shows, quizzes, lifestyle programs and news. For more information about this dataset, including a reference to pre-trained n-gram LMs, see (Verwimp et al., 2017a); for more details about the normalization, we refer the reader to (Verwimp et al., 2016). The vocabulary for both Dutch datasets is limited to the 100k most frequent words (the full vocabulary size is 145k for CGN and 330k for the subtitles dataset).

6. Experimental results

6.1. Perplexity

In table 2, we report perplexity results for the pre-trained models.

Data	5-gram	sentence LSTM	discourse LSTM
PTB	147.9	102.4	**84.1**
Wiki	231.0	150.6	**98.2**
CGN	395.2	257.6	**192.6**
Sub	114.5	74.4	**65.1**

Table 2: Test perplexities for the pre-trained sentence and discourse LSTM LMs compared to 5-gram LMs.

We compare a 5-gram model with interpolated modified Kneser-Ney smoothing (Chen and Goodman, 1999) trained with SRILM (Stolcke, 2002), with two baseline LSTM models, one trained on sentence-level batches and one trained on discourse-level batches. More details about the complexity of the models can be found on the Github repository and the download page. The perplexities for the English benchmark datasets are comparable to results for baseline word-level models reported in the literature: e.g. Jozefowicz et al. (2015) report a perplexity of 81.4 for a standard LSTM model on PTB and Grave et al. (2017) report a baseline perplexity of 99.3 for WikiText. As expected, discourse-level batching performs better than sentence-level batching: the improvement is largest for WikiText that has many long-term dependencies.

Input/output unit	Perplexity
word	84.1
character 2-gram	101.1
word-char	**83.6**

Table 3: Test perplexities for models trained with different input/output units on Penn TreeBank. The 'word-char' model is one to which 9 characters starting from the end of the word have been added.

In table 3, we report results for LMs trained with different input units. We see that models with concatenated word and character embeddings (last row) perform slightly better than word-level models, and that character 2-grams as input decreases the performance. Note that we did not do extensive experiments with character n-gram input, so it is very well possible that with a more thorough investigation, better results can be obtained. For example, this model might be interesting for other, more morphologically complex languages. Using a convolutional layer before the LSTM layer might also improve them. A character-level model with the

same hidden size, which has characters as input and output (and hence a different output vocabulary than the models in table 3), has a perplexity of 2.8.

6.2. Interpolation weights

The toolkit can also be used to calculate interpolation weights between different neural LMs and/or n-gram LMs. As an example, we generate a debugging file for the validation set of PTB with our pre-trained LSTM LM. If we generate a debugging file for a 5-gram LM with interpolated modified Kneser-Ney smoothing with the SRILM toolkit, we can automatically define the optimal interpolation weights on the validation set, which are 0.24 for the n-gram model and 0.76 for the LSTM model. These weights can subsequently be used in applications such as N-best rescoring. The perplexity of the interpolated model (see table 4) has improved 8% with respect to the LSTM LM alone.

Model	Valid Perplexity	Test Perplexity
5-gram	155.1	147.9
LSTM	107.5	102.4
interpolation	**98.6**	**94.7**

Table 4: Validation and test perplexities for a 5-gram LM, an LSTM LM and their interpolation on PTB.

6.3. Predicting next word(s)

In table 5, we show some examples of generating text with trained LMs. We show the difference between generating text with a model trained on PTB or on WikiText, and between generating the most likely word at every time step ('-p') or sampling based on a multinomial distribution ('-s'). Since the class of unknown words is treated as a word itself, unknown words can in principle be predicted too, but in the case of sampling from the multinomial distribution, we sample another word is the unknown word has been chosen. We observe that sampling from the multinomial distribution, hence not always choosing the most probable word, results in sequences that are more 'free' in the sense that they are often longer and contain less frequent words.

6.4. Rescoring

The rescoring option can be used to assign log probabilities to a list of sentences with a trained LM. For example, in table 6 we show some results after rescoring the 100-best lists from the DARPA WSJ'92 and WSJ'93 data sets, used by among others (Xu et al., 2009) and (Filimonov and Harper, 2009). The LM used for rescoring is the pre-trained PTB LM. Note that these log probabilities should be scaled by the number of words in the sentence (with a *word insertion penalty*) to properly compare them, since hypotheses with more words get a lower probability as more log probabilities are added. Compare for example the log probability for *he made a sales goal he says* (-31.997), and for the exact same sentence but with an extra word at the end, *he made a sales goal he says it* (-35.9978).

seed	**consumers may...**
PTB-p	be able to <unk> the <unk> of the <unk>
PTB-s	no longer be active
Wiki-p	be used to be a <unk>
Wiki-s	have made 0 - 3 victory in Mahwah
seed	**consumers may want...**
PTB-p	to be <unk>
PTB-s	to keep the gop golden share
Wiki-p	to be the first to be <unk>
Wiki-s	to have a strong impact on the storytelling
seed	**in recent...**
PTB-p	years
PTB-s	months after four years of investments
	last month for the year alone
Wiki-p	years
Wiki-s	years , broadcasts by Conservative Party theatre
	in the 18th century
seed	**The city 's growth has reflected the push and pull of many social...**
PTB-p	security benefits
PTB-s	states in the areas of southeast asia
Wiki-p	contexts
Wiki-s	forms

Table 5: Results of predicting the most probable sequence ('-p') or sampling based on a multinomial distribution ('-s') with the pre-trained PTB and Wiki models given several seeds extracted from the validation data from PTB and Wiki.

Hypothesis	Log prob
he made a sales goal he says	-31.9997
he made a sales coal he says	-32.0020
he made a sales goal he sets	-32.0021
he made a sales call he says	-32.0053
he made its sales goal he says	-32.0184
he made its sales coal he says	-32.0221
he made its sales call he says	32.0255
he made as sales goal he says	-32.0304
he made as sales coal he says	-32.0331
he made as sales call he says	-32.0360
he made it sails call he says	-32.0619
he may to a sales goal he says	-35.9857
he made a sales goal he says it	-35.9978
he made a sale of coal he says	-35.9994
he made a sales to call he says	-36.0034
he made a sales call he says it	-36.0043
he made a sale to call he says	-36.0083

Table 6: Sorted log probabilities for part of the WSJ N-best list development set, assigned by a model trained on PTB.

7. Conclusion

We release an open-source toolkit for language modeling based on TensorFlow: `https://github.com/lverwimp/tf-lm`. It contains several options for input/output unit, batching, training and testing and is easy to adapt. We show that it obtains state-of-the-art performance on English benchmarks, and release LMs trained on those benchmarks and on corpora of spoken Dutch.

8. Acknowledgements

This research is funded by the Flemish government agency IWT project 130041, SCATE.

9. Bibliographical References

Abadi, M., Agarwal, A., Barham, P., Brevdo, E., Chen, Z., Citro, C., Corrado, G. S., Davis, A., Dean, J., Devin, M., Ghemawat, S., Goodfellow, I., Andrew Harp, G. I., Isard, M., Jozefowicz, R., Jia, Y., Kaiser, L., Kudlur, M., Levenberg, J., Mané, D., Schuster, M., Monga, R., Moore, S., Murray, D., Olah, C., Shlens, J., Steiner, B., Sutskever, I., Talwar, K., Tucker, P., Vanhoucke, V., Vasudevan, V., Viégas, F., Vinyals, O., Warden, P., Wattenberg, M., Wicke, M., Yu, Y., and Zheng., X. (2015). Tensorflow: Large-scale machine learning on heterogeneous systems. *Software available from tensorflow.org*.

Chelba, C., Norouzi, M., and Bengio, S. (2017). N-gram language modeling using recurrent neural network estimation. Technical report, Google.

Chen, S. F. and Goodman, J. (1999). An empirical study of smoothing techniques for language modeling. *Computer Speech and Language*, 17:359–394.

Chollet, F. (2015). Keras. `https://github.com/fchollet/keras`.

Collobert, R., Kavukcuoglu, K., and Farabet, C. (2011). Torch7: A matlab-like environment for machine learning. In *BigLearn, NIPS Workshop*.

Enarvi, S. and Kurimo, M. (2016). TheanoLM – An Extensible Toolkit for Neural Network Language Modeling. In *Proceedings Interspeech*, pages 3052–3056.

Filimonov, D. and Harper, M. (2009). A joint language model with fine-grain syntactic tags. In *Proceedings Conference on Empirical Methods in Natural Language Processing (EMNLP)*.

Grave, E., Joulin, A., and Usunier, N. (2017). Improving neural language models with a continuous cache. In *Proceedings International Conference on Learning Representations (ICLR)*.

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

Jia, Y., Shelhamer, E., Donahue, J., Karayev, S., Long, J., Girshick, R., Guadarrama, S., and Darrell, T. (2014). Caffe: Convolutional architecture for fast feature embedding. *arXiv preprint arXiv:1408.5093*.

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

Kim, S. (2017). word-rnn-tensorflow. `https://github.com/hunkim/word-rnn-tensorflow`.

Marcus, M. P., Marcinkiewicz, M. A., and Santorini, B. (1993). Building a Large Annotated Corpus of English: the Penn Treebank. *Computational Linguistics*, 19:313–330.

Merity, S., Xiong, C., Bradbury, J., and Socher, R. (2016). Pointer sentinel mixture models. *arXiv preprint arXiv:1609.07843*.

Mikolov, T., Karafiát, M., Burget, L., Černocký, J., and Khudanpur, S. (2010). Recurrent neural network based language model. In *Proceedings Interspeech*, pages 1045–1048.

Mikolov, T., Kombrink, S., Deoras, A., Burget, L., and Černocký, J. (2014). RNNLM - recurrent neural network language modeling toolkit. In *Proceedings Interspeech*, pages 2093–2097.

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

Ozair, S. (2017). char-rnn-tensorflow. `https://github.com/sherjilozair/char-rnn-tensorflow`.

Pelemans, J., Shazeer, N., and Chelba, C. (2016). Sparse non-negative matrix language modeling. *Transactions of the Association for Computational Linguistics*, 4:329–342.

Schwenk, H. (2013). CSLM – a modular open-source continuous space language modeling toolkit. In *Proceedings Interspeech*, pages 1198–1202.

Stolcke, A. (2002). SRILM an extensible language modeling toolkit. In *Proceedings International Conference Spoken Language Processing*, pages 901–904.

Sundermeyer, M., Schlüter, R., and Ney, H. (2012). LSTM Neural Networks for Language Modeling. In *Proceedings Interspeech*, pages 1724–1734.

Sundermeyer, M., Schlüter, R., and Ney, H. (2014). rwthlm – The RWTH Aachen University Neural Network Language Modeling Toolkit. In *Proceedings Interspeech*, pages 2093–2097.

TensorFlow. (2017). Recurrent neural networks. `https://www.tensorflow.org/tutorials/recurrent`.

Theano Development Team. (2016). Theano: A Python framework for fast computation of mathematical expressions. *arXiv preprint arXiv:1605.02688*.

Verwimp, L., Desplanques, B., Demuynck, K., Pelemans, J., Lycke, M., and Wambacq, P. (2016). STON: Efficient subtitling in Dutch using state-of-the-art tools. In *Proceedings Interspeech*, pages 780–781.

Verwimp, L., Pelemans, J., Lycke, M., Van hamme, H., and Wambacq, P. (2017a). Language Models of Spoken Dutch. *arXiv preprint arXiv:1709.03759*.

Verwimp, L., Pelemans, J., Van hamme, H., and Wambacq, P. (2017b). Character-Word LSTM Language Models. In *Proceedings of the European Chapter of the Association for Computational Linguistics (EACL)*, pages 417–427.

Xu, P., Karakos, D., and Khudanpur, S. (2009). Self-supervised discriminative training of statistical language models. In *Proceedings IEEE Automatic Speech Recognition and Understanding Workshop (ASRU)*.