LANGUAGE MODELLING USING DEEP LEARNING STRUCTURES

Ahmet Baglan, Mehmet Eyyupoglu, Wisse Barkhof

Technical University of Denmark
Deep Learning-02456
repo: github.com/wissebarkhof/neural-language-modeling

ABSTRACT

In this project, RNN and LSTM modules were implemented using only primitive functionalities of the PyTorch library. The modules were used in the implementation and development of state-of-the-art Language Models using 4 different data sets. To prevent overfitting, many regularization methods including DropConnect, Weight Tying, Variable Backpropagation Length were applied to the models. This is a paper investigating the fundamentals of language modelling as well as models used for this field.

1. INTRODUCTION

Language modelling (LM) is the art of estimating the probability distribution of various linguistic components such as words and sentences. Lately, LM has become a more and more interesting research topic as it is fundamental to many advanced Natural Language Processing (NLP) task. Advanced LM models proved to be useful in many applications such as speech recognition [21], machine translation [22] and learning token embeddings [23].

Recently, deep learning has been driving most of the innovation in this field. Especially Recurrent Neural Networks (RNN) have been exploited more and more to obtain good performance on this task. In theory, RNN should be able to use information from any sequence length data but in practice the models are able to only remember a few steps back. This is because of a well known problem about RNN which is called vanishing gradient problem. As the sequence length gets bigger, the number of recurrent layers in RNN increases. In such a deep network, the gradient flowing back in back propagation gets smaller and smaller.

A popular derivative of RNNs are long short-term memory networks (LSTMs) which use a second memory cell to create long term connections as well. Thus, LSTMs mitigate the vanishing gradient problem [24], [25].

In this project, we have developed our own LSTM and RNN modules and implemented them for Language Modelling tasks by using many different regularization methods. Implemented regularization methods include DropConnect [13], Varying BPTT length, Embedding Dropout [16], Weight Tying [20] and L2 Regularization. The models developed was finally applied to different data sets to reach state of art level performance

The performance of language models was evaluated through a metric called perplexity, which is the average entropy for a word of the data. Perplexity is the uncertainty the model has in predicting some text.

The best model that we could develop in this project had a perplexity of 75.68 on the established dataset of Penn Treebank which is pretty close to the state-of-art,

2. RELATED WORK

In this section we will introduce, the Language Modelling literature including count based models and continuous space models.

2.1. Count Based Language Models

Statistical information about the language can be used to estimate the joint probability distribution of a sequence of words. One of the main examples of count based language models is the n-gram model. This requires making an n-th order Markov assumption and estimating n-gram probabilities.

In n-gram models, prediction of a whole sequence of words is divided into the prediction of a single word at a time. Thus the probability p(w1,w2,,wn) of a sequence is assumed as follows.

$$P(w_m|w_1,..,w_{m-1}) = P(w_m|w_{m-n},..,w_{m-1})$$

or equivalently

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

$$\prod_{i=1}^{m} P(w_i|w_{i-(n-1)},...,w_{i-1})$$

This is also a Markov chain. The basic idea for n-gram LM is that we can predict the probability of word by its predecessors. This is done by the following examples.

$$P(w_2|w_1) = \frac{count(w_1,w_2)}{count(w_1)}$$
$$P(w_3|w_1,w2) = \frac{count(w_1,w_2,w_3)}{count(w_1,w_2)} \quad (1)$$

However, one of the main problems with this approach is that, the joint probability of a sequence forming a sentence would be zero when calculated using a simple n-gram model if the same combination was not encountered in the training corpus. Thus most likely all the test cases which are not in the training set would give a zero probability. This is not a realistic assumption given that language is very flexible and out of sample test cases are most likely to occur. This problem is called as sparsity. There are various back-off [1] and smoothing techniques [2], [3], which would refer to a 3-gram if the 4-gram is not found, but no good solution exists.

Despite the smoothing techniques, sparsity is a great problem for LM. For example, for a 10 word sentence and 100.000 word vocabulary there are 10^{50} different permutation of sentence. The state-of-the-art by [1] used 140 GB RAM for 2.8 days to create a model on 126 billion tokens.

Another problem with n-gram method is that there is a window size and any relation between two word beyond this window is disregarded. Thus Markow assumption is a necessity but is a wrong assumption.

2.2. Continuous Space Based Models

One way to model a complex conditional probability function, is to approximate it with a deep neural network [17], which is in fact the state-of-the-art technique for language modelling, also known as neural language modelling (NLM). The most well known application of neural networks to a task related to language modelling, is Word2Vec [18].

Here, the task at hand is to create a meaningful, continuous vector representation for words. Since most machine learning algorithms require a fixed length numeric feature vectors as input, rather than strings, often bag-of-words was used to create a one-hot encoding of all the vocabulary. In case of a real world data, this can lead to very large, sparse vectors. So, using a word's n neighboring words -referred to as *context*, a feed forward neural network is used to predict the *center word*, given a context, for a large body of text. This is achieved by optimizing

$$P_\theta(w_i|c) = \frac{e^{u_i}}{\sum_{i=1}^{V} e^{u_i}} \quad where \quad u_i = v_{w_i}.v_c \quad (2)$$

where v_c is the corresponding pairs of context words matched with our center word w_i.

This way, one-hot encoded vectors are converted into a lower dimension which is referred to as an *embedding*. Since we are predicting a word from surrounding words, this is a similar task to language modelling, but we do not take into account the word order and use words both before and after the predicted word. However, language modelling does often make use of a word-embedding, which can be defined explicitly or learned as a part of the model training, leading to better results [12].

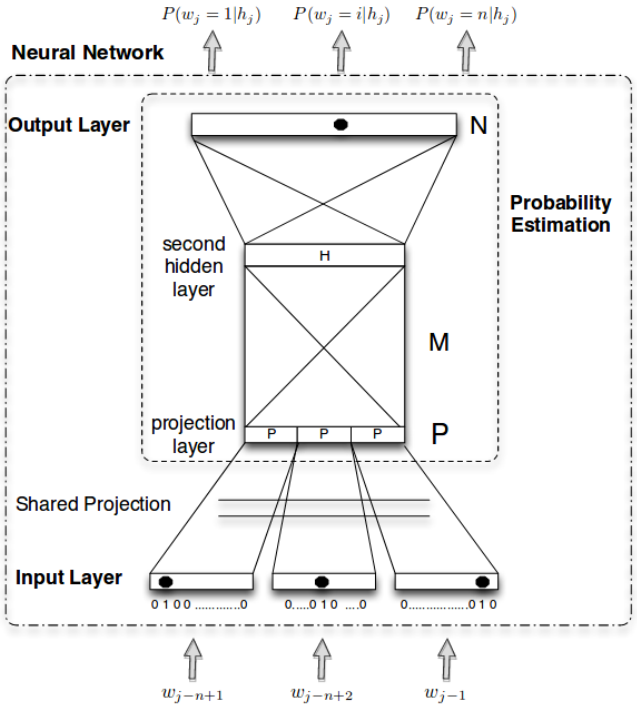

Fig. 1: The neural network language model architecture. P, N and H are the size of one projection, one hidden layer and the output layer respectively. h_j denotes the context $w_{j-(n-1)}^{j-1}$, the image is taken from [28]

Learning meaningful word embedding using NLM, words are projected to the continuous space during the training. For example, [27] a neural network model that jointly learns the word embeddings and probability distribution of the next word in the input corpus. The basic structure of this network can be seen at Figure 1.

Currently state of the are for NLM are recurrent neural network based models. RNN structures such as [5] and more specifiaclly long short-term memory based models and similar variants such as GRU are generally exploited for obtaining modern LM structures.

3. METHODS

3.1. Recurrent Neural Networks

A large downside of using feed-forward neural networks to do language modelling is that the size of the window needs to be predetermined in advance, and thus a recurrent neural

network (RNN) was proposed [5]. This neural network architecture uses an internal hidden state h_t at every step t in a sequence to incorporate structure from sequential data and predict $\hat{y}_t$ from x_t. In this case, $x_1, \ldots, x_{t-1}, x_t$ are word vectors, which can be one-hot encoded or one of the word embeddings mentioned before. The input x_t, hidden state h_t and output $\hat{y}_t$ are computed as follows:

$$h_t^j = f(W^{(hh)} h_{t-1} + W^{(hx)} x_t) \tag{3}$$

$$\hat{y}_t^k = g(W(y) h_t) \tag{4}$$

Where $f(z)$ is the activation function, for example a *sigmoid* function:

$$f(z) = \frac{1}{1 + \exp^{-z}} \tag{5}$$

and $g(z)$ is the *softmax* function:

$$g(z_m) = \frac{\exp z_m}{\sum_k \exp z_k} \tag{6}$$

The matrices $W^{(hh)}$, $W^{(hx)}$ and $W^{(y)}$ are the same throughout the time steps and they store the weights that are updated through backpropagation. Their superscripts indicate what state they modulate. The equations are taken from [5] but adapted in order to keep notation consistent.

In figure 2 a schematic overview is given of how the different equations influence each other in a simple RNN, sometimes called an Elman net. A practical problem with simple RNN structures is called the *vanishing/exploding gradient problem* sak2014. The issue here is that as the gradient of the error is propagated back many steps in the sequence of the data, the gradient tends to either blow up or completely vanish due the the consecutive power transformations on this gradient. This means that in reality it becomes hard to actually update a model for more than 5 - 10 words in the past, even though this happens regularly in natural language. One way to deal with this is with an adaption to the architecture called the Long-Short Term Memory model.

3.2. LSTM

The Long-Short Term Memory (LSTM) network architecture was introduced in 1997 by Hochreiter [6] and extends the idea of an RNN by using special memory cells as a way to include updates over large time lags. These memory cells store the long term dependencies within the sequence and come in two forms, a *new memory cell* $\tilde{c}_t$ and a *final memory cell* c_t. Additionally, there are 3 vectors that influence how much information of each of these cells flows from one state to another, the *input gate* i_t, *forget gate* f_t and the *output gate* o_t.

The mathematical formulation of the LSTM, taken from [10], are as follows:

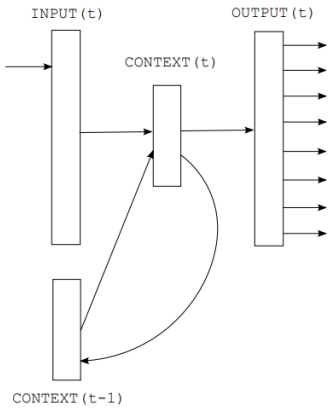

Fig. 2: A schematic overview of a simple recurrent neural network, taken from [5]

.

$$i_t = \sigma(W^i x_t + U^i h_{t-1}) \quad \text{Input Gate} \tag{7}$$

$$f_t = \sigma(W^f x_t + U^f h_{t-1}) \quad \text{Forget Gate} \tag{8}$$

$$o_t = \sigma(W^o x_t + U^o h_{t-1}) \quad \text{Output Gate} \tag{9}$$

$$\tilde{c_t} = tanh(W^c x_t + U^c h_{t-1}) \quad \text{New Memory Cell} \tag{10}$$

$$c_t = i_t \circ \tilde{c_t} + f_t \circ \tilde{c}_{t-1} \quad \text{Final Memory Cell} \tag{11}$$

$$h_t = o_t \circ tanh(c_t) \quad \text{Output} \tag{12}$$

Where σ is the sigmoid function (see equation 5), $[W^i$, W^f, W^o, W^c, U^i, U^f, U^o, $U^c]$ are weight matrices that are learned by the model, and $\circ$ is the element-wise multiplication of two matrices.

The gates i_t, f_t and o_t all use a sigmoid 5 function to get to a number of 0 and 1, which at any given time t determines how much of the input or previous states should be incorporated in the next state. The input gate does this the information from the *new memory cell* $\tilde{c}_t$ that is created from the input at t and the previous state h_{t-1} to the *final memory* c_t. The forget gate does this for the previous step $\tilde{c}_{t-1}$, and the output gate blocks or unblocks the flow of information the final memory $c(t)$ to the output $h(t)$.

For simplicity, the equations for the RNN and LSTM have been given for a single layer, but they are easily extendable to a multilayer network, where every hidden state h_t receives information from both the previous time step $t-1$ as from the previous layer $l-1$ at the same time step t. Here, $l \in L$, where L is the number of layers. [9] write this as follows:

$$\text{LSTM:} h^{l-1}(t), h^l(t-l), c^l(t-l) hl(t), c^l(t) \tag{13}$$

3.3. State-of-the-Art

One downside of using neural networks in general and LSTMs in particular, is that a lot of parameters are intro-

duced, which are prone to overfit. Achieve generalizing performance often relies on being able to regularize the network sufficiently [10]. [7] did a controlled comparison of the state-of-the-art language models, where they found that [10] perform best with a 57.3 perplexity on the Penn Tree Bank [11]. This project therefore implements some of the techniques applied in their research, with the aim of achieving a comparable perplexity. We applied regular dropout to the encoder layer and L2 regularization on all the model parameters, as is standard practice. Some more inventive regularization techniques from Merity at al. will be discussed below.

3.3.1. DropConnect

The result achieved by Merity et al. [10] depends heavily on a the use of DropConnect, introduced by Wan et al. in 2013 [19], which applies dropout on the hidden-to-hidden recurrent weights connections in the LSTM. Most other recurrent dropout attempts, have been focused on applying dropout to the hidden layer h_t between timesteps or on the update to the memory cell c_t, which prevents the use of using exisiting LSTM implementations. With DropConnect, the same weights are dropped for the whole forward pass on a sequence of data and thus it can be applied without changing the the internal workings of the LSTM. This allows for easy application to any LSTM implementation, whether it is NIVIDIA cuDNN or a naive implementation, like in our case.

3.3.2. Variable backpropagation length

To improve data efficiency further, variable length backpropagation sequences were used. During training, the full sequence of text is broken up in shorter sequences to backpropagate over. Often, this is done by experimentally determining a fixed, optimal sequence length. However, this leads to a splitting up the data in the same way every time and since the first word in the sequence has nothing to backpropogate to, a part of the data is never used. Merity et al. propose the use of a variable sequence length, which is normally distributed with some variance around a set mean. This way, the sequence length can still be set to be in some range, but all connections are learned.

3.3.3. Weight tying

An easy way to decrease the total number of learned parameters, is by forcing the weights of the encoder and decoder to be equal. Weight tying has a theoretical motivation and prevents the model from having to learn a one-to-one correspondence between input and output inan2016.

4. EXPERIMENTAL SETUP

4.1. Data

4.1.1. Penn Tree bank

The model is trained on the Penn Tree bank [11], which is a large corpus which consists of 929k training words, 73k validation words, and 82k test words. It has 10k words in its vocabulary. This corpus functions as a benchmark set in many current language modeling papers [7], [10], so it is a good way to compare this study's work to related studies.

4.1.2. Holy Scripture

Holy books might also be interesting in terms of application of the model. Therefore, trained and tested with Quran and Bible.

In order to keep the preprocessing of the data easy, the split was done arbitrarily as %80 for training, %10 for validation and %10 for test data. This decision is more convenient since it is not really possible to get a homogenous split. An alternative solution would be splitting the text sentence by sentence and randomizing them. However, in this case this would lead to an interruption of the context.

4.1.3. Friends

For the sake of creating LM on different dataset, scripts from the famous tv series 'Friends' was collected. At [26] you can find all the scripts collected.

The scripts were first collected and around 80% first words were used as training set, next 10% were used as validation and final 10% were used as test. Thus more or less first 8 seasons were used as training, season 9 was used as validation and the last season was used for testing. Note that one major drawback of this split is that at each season the main topic of the series change. So, using one season as test is a bit harsh but a convenient way to split the data.

The size of the corpus is 872k training words, 110k validation words and 121k test words. The vocabulary size of the dataset was 28k.

Note that very little preprocessing was applied on the scripts. Only multi white spaces were converted to single space.

4.2. Architecture

The network that was used consisted of an one linear embedding layer as encoder, then a multi-layer LSTM with dropped weights according to the DropConnect procedure where the number of layers has been tweaked as a hyper parameter, and then a decoder layer. Training was done using only vanilla gradient descent. Since the aim was to build as much as we could ourselves, we updated the parameters 'manually' for every batch. Also, L2 regularization was applied 'manually'

by adding a term to the loss. The hidden layers were initialized with values from uniform distribution on $[-\sqrt{H}, \sqrt{H}]$ where H is the number of units in the hidden layer. The encoder and decoder layers were initialized with values from a uniform distribution on $[-0.1, 0.1]$. In figure 3 the main architecture and experimental setup is shown in a schematic fashion.

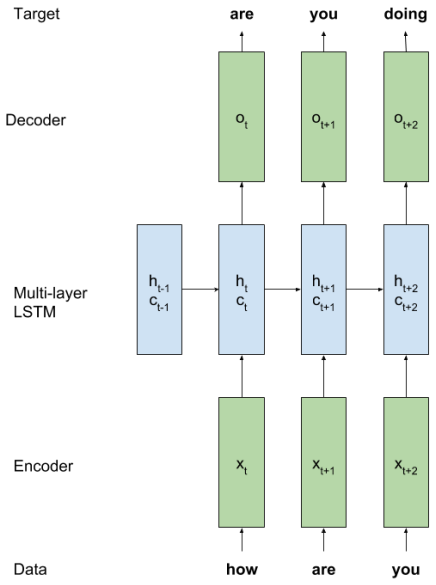

Fig. 3: Experimental architecture

A search for optimal hyper parameters was conducted in a trial and error type of fashion and not in an extended grid search. If something worked well, those settings were kept for next experiment. The hyper parameters that were experimented with were:

- number of LSTM layers
- size of encoding
- size of hidden layer
- BPTT sequence length mean
- batch size
- weight tying
- encoder dropout level
- DropConnect dropout level
- l2 regularization

4.3. Technological Specification

For the experiments, Google Colab and its Tesla K80 graphics card with 11,5 GB ram was used.

5. EXPERIMENTAL ANALYSIS

5.1. Quantitative Result

In table 1, the lowest achieved perplexities for the custom LSTM and the PyTorch implementation (which uses the NIVIDIA cuDNN in the backend) on the different datasets are reported after 40 epochs. The PyTorch implementation was only trained on the PBT dataset as a benchmark, to ensure the implementation was correct. For this dataset, the PyTorch implementation slightly outperforms the custom built LSTM in both validation as test perplexity. However, if we consider the training time, the PyTorch implementation is almost 10x faster, as can be seen in figure 4.
On the PTB, the best results were achieved with a 3 layer network, with an embedding size of 400 as well as a 400 hidden units. This way, weight tying could be applied, since in and output need to be equal dimensions. A batch size of 20 and a variable sequence length with mean 70 was used.
On the Friends dataset, the same model using DropConnect (0.5 probability) with weight tying returned higher perplexities. This is still probably due to the fact that the test and validation set for this data have slightly different context compared to training data.
Quite interesting to see is that in all cases the test perplexity is lower than the validation perplexity and that the difference between the custom LSTM and PyTorch implementation is about equal for all datasets.

Dataset	***PyTorch***		***Custom***	
	Validation	Test	Validation	Test
PTB	75.58	71.36	73.46	69.71
Bible	73.7	—	76.1	—
Quran	91.0	—	95.6	—
Friends	108.2	106.5	112.1	110.8

Table 1: Validation and test perplexity scores for the custom build LSTM and PyTorch implementation for the different datasets after 40 epochs.

5.2. Qualitative Result

In figure 5 there are 2 examples of generated text outputs given a starting word. They consist of some words that are predicted given the previous word/words and the hidden units. In order to increase the variation of the predictions, probability distribution of a next word is scaled up by a number called, temperature. This leads us to have bigger difference between the word probabilities. This can be seen in figure 5. This is helpful especially when the model is too confident about a next word since generated sequence may get in a loop.

The text that is obtained with a higher temperature (Figure 6-a) is compared with a low temperature(Figure 6-b). Result

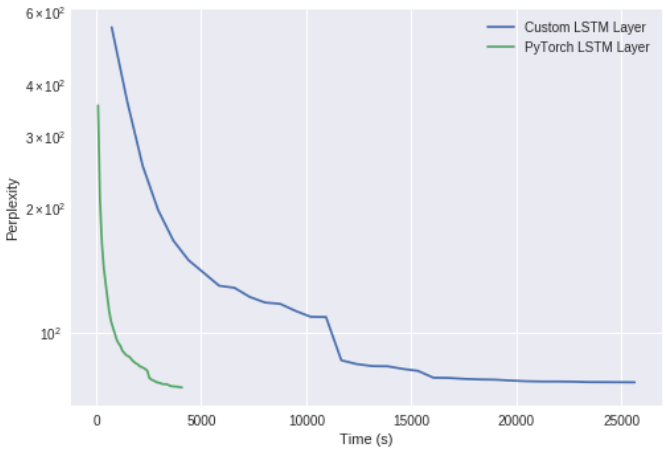

Fig. 4: Time comparison of this project's LSTM implementation and the PyTorch implemntation, which uses NIVIDIA cuDNN, for 40 epochs on PBT.

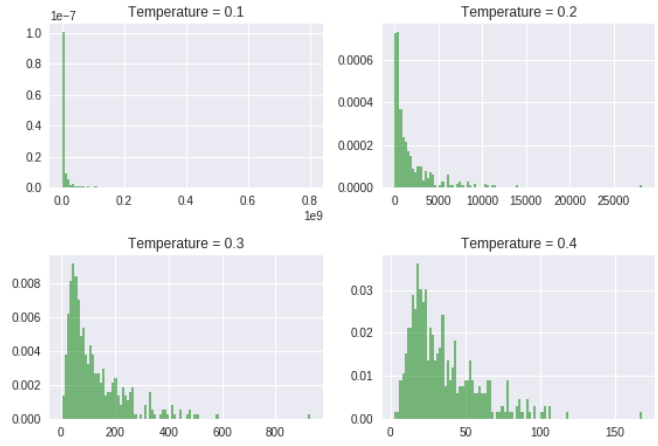

Fig. 5: Predicted distributions of the model with different temperatures are shown. When temperature is low, there will be a few words that are likely to be chosen

obtained from figure 5 matches with the qualitative result obtained from figure 6. With high temperature while we get high variety of words the words get less meaningful.

6. DISCUSSION

Even though the custom LSTM got a perplexity score that was close to the score achieved by the PyTorch implementation, it is still strange they are not exactly equal. In these kind of tasks, a difference of 2 in perplexity can still be significant. One explanation could be that it is to do with randomized initialization and that it is due to circumstances, but it seems unlikely that effect would be this large. Further, the naive custom LSTM layer was an order of 10 times slower than the PyTorch implementation, but this was not unexpected since the

negotiations such improved arms p.m form decided meaning available rouge medicine testimony analyzing additional fortunately high signal tour ballot weekly last evil uncertainties nights asia visible maximum feeling hazard boys testify crimes journalist slip campus which pleased forget management salesman stabilize sixth hardware rate weekly enjoy conducting throw neutrons apart

(a) temperature is 6

by a major party are the market was one of a new york stocks are expected to $ 300-a-share a $ 300-a-share co. is seeking a stock in the latest quarter included a executive said the walt disney closed at a $ half as many of a major response to begin trading on the first group said long-term investment trust said its outstanding figures have been elected chairman and the board of of the units of a $ 300-a-share of $ 300-a-share corp. a $ 300-a-share that had been a $ 300-a-share $ 300-a-share & drew the company 's vice

(b) temperature is 1

Fig. 6: Comparision of the 2 different sampling. [29]

cuDNN backend is highly optimized for running on GPUs.

The true state-of-the-art perplexity reported by Merity et al. [10] on the PTB, being 60 for validation and 57.3 for test, was also not achieved. This is in large part due to the fact that only vanilla gradient descent was used and not stochastic gradient descent or non monotonic average stochastic gradient descent (NT-ASGD) like proposed in the paper. During training, the loss leveled out after 30 or 40 epochs, which indicates the need for a more sophisticated optimizer. On top of that, we only implemented a subset of all the regularization techniques proposed.

7. CONCLUSION

In this project we have successfully developed a language model using the RNN and LSTM modules that we have implemented. From Moreover, we have used some of the regularization methods introduced by [10] Even though we had some hardware limitations, time constraints, a perplexity of 75.68 was reached. The model was applied to several different datasets and structure-wise logical text was generated.

8. REFERENCES

[1] R. Kneser and H. Ney. Improved backing-off for n-gram language modeling. In International Conference on Acoustics, Speech and Signal Processing, pages 181184, 1995.

[2] S.F. Chen and J.T. Goodman. An empirical study of smoothing techniques for language modeling. Computer, Speech and Language, 13(4):359393, 1999.

[3] J .Goodman. A bit of progress in language modeling. Technical Report MSR-TR-200172, Microsoft Research, 2001. [4] Yoshua Bengio, Rejean Ducharme, Pascal Vincent, and Christian Jauvin. A neural probabilistic language model. Journal of Machine Learning Research, 3:11371155, 2003.

[4] Tom Young, Devamanyu Hazarika, Soujanya Poria, Erik Cambria.(2017). Recent Trends in Deep Learning Based Natural Language Processing. arXiv preprint arXiv:1708.02709

[5] Mikolov, T., Karafit, M., Burget, L., Cernock, J., Khudanpur, S. (2010). Recurrent neural network based language model. INTERSPEECH.

[6] Hochreiter, S., Schmidhuber, J. (1997). Long Short-Term Memory. Neural Computation, 9, 1735-1780.

[7] Melis, G., Dyer, C., Blunsom, P. (2017). On the state of the art of evaluation in neural language models. arXiv preprint arXiv:1707.05589.

[8] Sak, H., Senior, A.W., Beaufays, F. (2014). Long Short-Term Memory Based Recurrent Neural Network Architectures for Large Vocabulary Speech Recognition. CoRR, abs/1402.1128.

[9] Zaremba, W., Sutskever, I., Vinyals, O. (2014). Recurrent Neural Network Regularization. https://arxiv.org/abs/1409.2329

[10] Merity, S., Keskar, N. S., Socher, R., (2017). Regularizing and Optimizing LSTM Language Models

[11] Marcus, M. P., Marcinkiewicz, M. A., Santorini, B.(1993). Building a large annotated corpus of english: The penn treebank. Computational linguistics, 19(2):313330

[12] Deep contextualized word representations. (2018). Peters, M. E., Neumann, M., Iyyer, M., Gardner, M., Clark, C., Lee, K., Zettlemoyer, L.

[13] Zaremba, W., Sutskever, I., and Vinyals, O. Recurrent neural network regularization. arXiv preprint,arXiv:1409.2329, 2014.

[14] Sutskever, I., Martens, J., Dahl, G., and Hinton, G. On the importance of initialization and momentum in deep learning. In International conference on machine learning, pp. 11391147, 2013.

[15] Mandt, S., Hoffman, M. D., and Blei, D. M. Stochastic gradient descent as approximate bayesian inference. arXivpreprint arXiv:1704.04289, 2017

[16] Gal, Y. and Ghahramani, Z. A theoretically grounded application of dropout in recurrent neural networks. In NIPS, 2016.

[17] Nielsen, A. M., "Neural Networks and Deep Learning", Determination Press, 2015

[18] Le, Q. V., Mikolov, T. Efficient Estimation of Word Representations in Vector Space, 2013

[19] Wan, L., Zeiler, M., Zhang, S., LeCun, Y, and Fergus, R. Regularization of neural networks using dropconnect. In Proceedings of the 30th international conference on machine learning (ICML-13), pp. 10581066, 2013.

[20] Inan, H., Khosravi, K., and Socher, R. Tying Word Vectors and Word Classifiers: A Loss Framework for Language Modeling. arXiv preprint arXiv:1611.01462, 2016.

[21] Yu, D. and Deng, L. Automatic speech recognition: A deep learning approach. Springer, 2014.

[22] Koehn, P. Statistical Machine Translation. Cambridge University Press, 2009

[23] P. Matthew, M. Neumann, M. Iyyer M. Gardner C. Clark K.Lee L. Zettlemoyer. Deep contextualized word representations.OpenReview, 2017

[24] Sepp Hochreiter. The vanishing gradient problem during learning recurrent neural nets and problemsolutions. International Journal of Uncertainty, Fuzziness and Knowledge-Based Systems, 6(02):107116, 1998.

[25] Yoshua Bengio, Patrice Simard, and Paolo Frasconi. Learning long-term dependencies with gradient descent is difficult. IEEE transactions on neural networks, 5(2):157166, 1994.

[26] https://fangj.github.io/friends/

[27] Y. Bengio, R. Ducharme, P. Vincent, and C. Janvin, A neural probabilistic language model, J. Mach. Learn.Res., vol. 3, pp. 11371155, Mar. 2003. [Online]. Available: http://dl.acm.org/citation.cfm?id=944919.944966

[28] Improving Continuous Space Language Models using Auxiliary Features(2015), Walid Aransa, Holger Schwenk, Loic Barrault

[29] https://github.com/mikkelbrusen/text-weight-visualizer