# N-gram and LSTM based Language Models

Zihao Wang

Research School of Computer Science,
the Australian National University
u6137905@anu.edu.au

**Abstract.** we constructed an n-gram model and a LSTM in building a word generating model. To predict words, the model relies on the context of input sequence so that the choice of an unseen word is conditioned on the history. This problem's character perfectly fits the strength of LSTM. LSTM language model (LM), compared to n-gram model, can predict a new word with respect to much longer history input, long term dependency, in other words. To take longer history in consideration, more unique words need to be stored. So that I chose to replace one-hot encoding by word embedding, which is more meaningful in language processing and memory efficient as well. Using word vector as targets in training are not benefitted much for a short training set but showed decent performance with the increment of vocabulary size. I use three groups of data for experiment which vary in size and learning difficulty. Resulting from my experiments, this word embedding encoding can fit in both n-gram LM and LSTM model. There is no better one in these two models according to my experiment but one performs better to a certain task and the other has its own advantages as well. In the experiment, n-gram LM is easier to train due to the limit number of parameters while LSTM requires carefully selected hyper parameters but converge faster. More specifically analyse will be discussed in section 5. All in all, LSTM model is more preferable for its stronger universality.

**Keywords:** language model, LSTM, word embedding, n-gram

## 1 Introduction

### 1.1 Motivation

Sequential data is a common form information. Literature, music and stock history all are sequential data. Therefore, sequential data prediction is a popular topic in machine learning and artificial intelligence research. It worth noticing that sequential data is produced from different probabilistic condition with respect to various of area where sequential data is drawn. For instance, music and text are both sequential data, mathematically, but they are under different assumptions. In music, the note is chosen almost freely though there are loose rules and conventions. In the contrary, from the aspect of text, the grammar is strict and some phrases are fixed combination of words such like 'give' and 'up'. The nature of text or language make the task of prediction more feasible and many studies in this area are proved success, which are introduced in section 2. Considering these, we compare two popular models in language prediction and give a conclusion on the strength and drawbacks of each model in section 6.

### 1.2 Preliminary Ideas

In this paper, we focus on text prediction and LM construction. In a statistical LM, we estimate the probability of a sequence of words denoting $W$ given history. It is obviously that each word in a sentence is not drawn i.i.d (independent and identically distributed) since words are conditioned on the previously seen sequence. Given this, the probability of an unseen word can be evaluated by

$$p(W) = \prod_{k=1}^{n} p(w_k \mid w_1, w_2, \ldots, w_{k-1}) , \quad (1)$$

according to the product rule. In practical, assume that the choice of each word only depends on a fixed length of context so that it satisfied Markov assumption

$$p(w_k \mid w_1^{k-1}) = p(w_k \mid w_{k-l}^{k-1}) , \quad (2)$$

where $w_1^{k-1}$ denotes the sequence $[w_1, w_2, \ldots, w_{k-1}]$ and $l$ is the length of the prefix that matters. Statistically, the number of parameters grows exponentially with respect to $l$. In the other hand, the LM model results in better accuracy with longer context in theory. This dilemma will be discussed in detail for the two models in this paper, respectively. For now, applying *eq. 2* to *eq. 1* we derive that

$$p(W_{l+1}^{n}) = \prod_{k=l+1}^{n} p(w_k \mid w_{k-l}^{k-1}) \quad (3)$$

### 1.3 Data Pre-processing

The learning model hardly improve if we feed words in directly. Some processing is applied on the context $w_1^{k-1}$, and we denote this mapping by $\psi(w_{k-l}^{k-1})$. The following equation stands for the same meaning in statistical but with modification as *eq.4*.

$$p_\theta(W_{l+1}^{n}) = \prod_{k=l+1}^{n} p_\theta(w_k \mid \psi(w_{k-l}^{k-1})) \ , \quad (4)$$

where $\theta$ is the parameter controls the distribution. We do not use word as a string for input directly, typically, each word was encoded in a one-hot style. Assume the vocabulary size is $n$, which indicates the number of unique word in this training set, the encoding of each word required a $n$ dimension vector where only one position is 1 and others were all 0 such as $[0, 0, \ldots, 1, 0, 0]$ meaning that the current word is the third last word in the vocabulary. The one-hot encode represents a sharp distribution (0 or 1) of $p(w_k)$ but the drawback of this method is being memory expensive. Additionally, one-hot encoding cannot show the relationship between each word. For example, 'cat' and 'kitty' are alike in some way but in one-hot encode there was no relation between these two. Thus, we replace the one-hot encoding by word embedding which is soft on distribution and in the contrary to one-hot encoding, word embedding is able to preserve the similarity of different words. Presenting by vectors, the distance of 'cat' in vector $a$ and 'kitty' in vector $b$ is computed as $\| a - b \|$ and the result would be small. For this reason, we pre-process training data by word embedding and use the notation $\psi(w_{k-l}^{k-1})$ to show that.

### 1.4 Statistical LM

To wrap the assumptions and ideas up, we introduce some general formula to evaluate LM. The language model aims to give probability of unseen words; therefore, the property is greater or equal to zero

$$p_\theta(w_k \mid \psi(w_{k-l}^{k-1})) \geq 0, \forall w_k \in W \ . \quad (5)$$

And we use cross-entropy loss in training

$$J_\theta(W_{l+1}^{n}) = -\sum_{k=l+1}^{n} p(w_k) \log p_\theta(w_k \mid \psi(w_{k-l}^{k-1})) \ , \quad (6)$$

which is also used as a model comparison criterion due to the purpose of this paper. Minimization of the loss function can be formally written as

$$\theta^* = \arg\min_\theta J_\theta(\widetilde{w}) \ . \quad (7)$$

To be clear, there are two main tasks in this project, one is to find the appropriate $\psi(x)$ and the other one is to appropriately estimate $p_\theta(w_k \mid \psi(w_{k-l}^{k-1}))$. Since we have stated the reason for choosing a particular $\psi(x)$, two models were introduced in the following as estimation methods.

## 2 Background and Related Work

Feed-forward neural network has the capability of estimating posterior probability. Although it only considers a fixed context length and the length is fairly short, the performance on word prediction is considerably improved since the first come out concept. We will take a n-gram feed-forward network as an example in this paper. On the other hand, recurrent neural network (RNN) has the property that predictive distribution does not dependent on a fix length context explicitly. Therefore, RNN considers longer context when making prediction. Due to the gradient vanishing problem [1], LSTM is proposed as an answer for long memory. Considering these, the predictive performance of n-gram LM and LSTM LM are interesting and valuable in sequential modelling.

The statistical language model was proposed long ago, S. Katz introduced a nonlinear recursive procedure to solve n-gram language prediction in 1987 [2] and Stolcke proposed a language model via a particular approach naming Bayesian Learning [3]. Later on, Yoshua et. al derived a distributed representation for words that improve the n-gram model [4]. In [5] and [6], great feed-forward networks were constructed on LM. More recently, M. Sundermeyer et al. [7] proposed an enhanced LSTM and an RNN model is designed by Mikolov et. al in 2010 [8].

## 3 Model Description

We introduce the principles and derive n-gram LM and LSTM with statistical evidence in this section. Specific settings and parameters will be explained in section 4.

### 3.1 N-gram

N-gram is so popular that lots of related research have focused on predicting words through it. In a n-gram model, only the previous $n-1$words are conditioned on when making a prediction, noting that $n$ denotes the length of context here rather than the total number of words in data. A typical n-gram model is trigram, which separated training set into 3-word groups where the first two words were the evidence for predicting the last word. For example, we had a trigram training group as $[(w_{i-2}, w_{i-1}), w_i]$ corresponding to the following equations. Inputs were one-hot encoded in the equations below but compatible with word embedding technique.

$$y_i = A_1\widehat{w}_{i-2} \circ A_1\widehat{w}_{i-1} \tag{8}$$

$$z_i = \sigma(A_2 y_i) \tag{9}$$

$$p(c(w_i)|w_{i-2}, w_{i-1}) = \varphi(A_3 z_i)\,|_{c(w_i)} \tag{10}$$

$$p(w_i|c(w_i), w_{i-2}, w_{i-1}) = \varphi(A_{4,\,c(w_i)} z_i)|_{w_i} \tag{11}$$

, where $\circ$ denotes concatenated, $\sigma$ denotes sigmoid function, $\varphi$ denotes softmax function and $c$ denotes the class that $w_i$ belongs to. By $A_1, A_2, A_3, A_4$, we denote the weight matrices of the network [9]. Specific activation function in *eq.9 - eq.11* can varies in practice. The choice in our experiment will be justified in section 4.

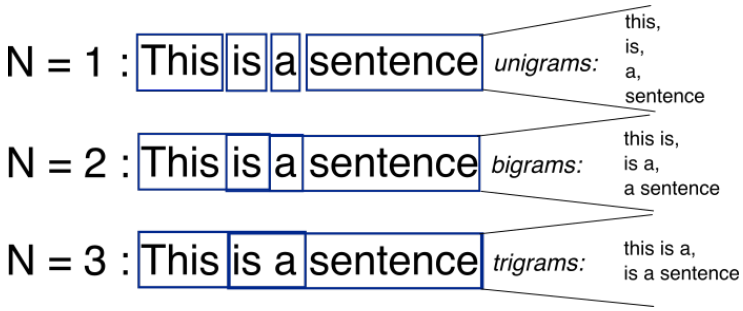


**Fig. 1.** sequence split for unigram, bigram and trigram [10].

And we use the first element in weight matrices as bias by convention and did not show them explicitly in the equations. The result $p(w_i|c(w_i), w_{i-2}, w_{i-1})$ in *eq.10* is normalized by softmax function. Adapting to our evaluation, we take $\theta = [A_1, A_2, A_3, A_4], p(w_i|c(w_i), w_{i-2}, w_{i-1})$ into the objective function $J_\theta(\widetilde{w})$ in *eq.6* for training. Here is one method to improve the performance of n-gram algorithm. By applying a perplexity-based prior distribution rather than use simply a uniform distribution of all word classes, the training cost was reduced strongly [9].

### 3.2 LSTM

LSTM is an elegant method for sequence generation and have bunches of advantages than traditional RNN, such as forget gate and the consideration of long dependency. To follow conventions, we need to introduce notations again. Let a sequence $\tilde{x} = [x_0, x_1, \ldots, x_n]$ as input and initialize cell states $h_0 = 0, c_0 = 0$, a standard LSTM compute an output $\tilde{y} = [y_0, y_1, \ldots, y_n]$ by taking one $x_i$ per time and iterating over time steps with the following equations

$$i_t = \sigma(W_{xi}x_t + W_{hi}h_{t-1} + W_{ci}c_{t-1} + b_i) \tag{12}$$

$$f_t = \sigma(W_{xf}x_t + W_{hf}h_{t-1} + W_{cf}c_{t-1} + b_f) \tag{13}$$

$$c_t = f_t c_{t-1} + i_t tanh(W_{xc}x_t + W_{hc}h_{t-1} + b_c) \tag{14}$$

$$o_t = \sigma(W_{xo}x_t + W_{ho}h_{t-1} + W_{co}c_{t-1} + b_o) \tag{15}$$

$$h_t = o_t \tanh(c_t) \tag{16}$$

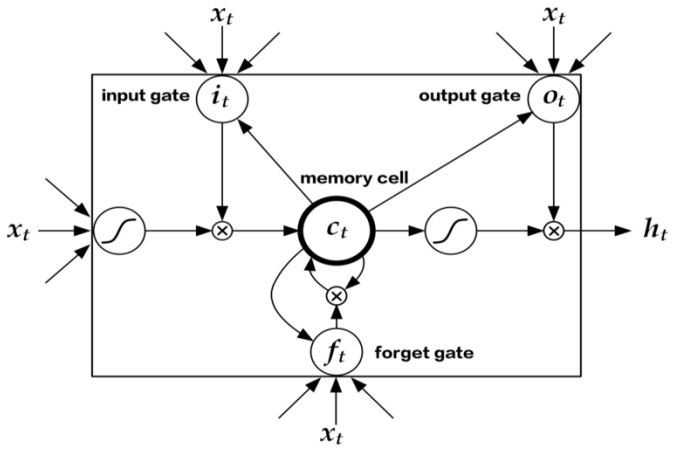


**Fig. 2.** LSTM cells [11].

Noting that we use $x$ to denote input and $W$ to denote weights, which is different from section 3.1. With these intermediate steps adding, LSTM is allowed to solve the gradient vanishing problem. In *eq.16*, we choose $\tanh(x)$ as the activation function and word distribution is gained by the elementwise multiplication between output state $o_t$ and the tanh activation function.

As to inputs, by applying word embedding as mentioned above, the dimension of vector for each word is defined on a constant so that the space it takes is fixed rather than linear to the size of vocabulary. On the output, we apply this trick again but reversely and got the predicted word from vector.

## 4 Method

To evaluate the n-gram and LSTM models, as well as how their performance change with respect to improvement thoughts, we design several word-generating tasks. Because of the limits of hardware for experiments, we will interrupt experiments which is severely slow regardless the theoretically high accuracy.

### 4.1 dataset details

We choose text that meaningful and grammar-correct. Our datasets are a snippet of Shakespeare sonnet, a medium long vernacular text and the whole article of 'Anna Karenina'. The selected part of Shakespeare sonnet (training set A) contains 115 characters and carefully cleaned, removed punctuation and irrelevant noise. This is a very common training set of n-gram model and we chose it because of its appropriate vocabulary size. The medium long text (training set B) contains 128533 characters but all basic and simple English words with only 67 unique characters in which. 'Anna Karenina' was a great literature (training set C), as you can imagine, contains 352928 words (not characters) in total and 29230 unique ones. This data set was also clean text without any noise.

We did not compare one-hot encoding with word embedding but apply word embedding as input processing for all experiments so that we did not have to use a fixed vocabulary set for fair. Instead, we set vocabulary specific for each training set for better performance and use the same encoding dimension to each word.

### 4.2 n-gram algorithm

We read training set A, $\tilde{x}^A$, as a string and split them to separated words. Next is to encode each unique word, labelling word with different number for the sake of word embedding. In PyTorch, we can call 'nn.embedding' for automatically word embed and store the return values for further processing. If we implemented a trigram model, a trigram list was built as $[(w_{i-2}, w_{i-1}), w_i]$. We train this model by feeding input left to right, applying linear operation in the first layer (not counting input layer). We modified the sigmoid function to $ReLU$ to transform first layer's output and feed the result in another linear layer. At last, applying $softmax$ function and calculate the log probability to get the final output. This n-

gram model has the extensibility with any appropriate $n$. Taking the advantage of that, we compared different $n$ in n-gram and the influence it had.

**Pseudocode for N-gram LM.**

split input to list of word: $W = [w_1, w_2, \ldots, w_k]$
set of unique words: $vocab = set(W)$
labeling word: $\{(\text{word: index})\}, \forall word \in vocab$
n-gram set: $((w_i, \ldots, w_{i+n-1}), w_{i+n}), \forall 0 < i < k$

**for** i in range **do**
    input and target: $(x_i, t_i) = ((w_i, \ldots, w_{i+n-1}), w_{i+n})$
    output: $y_i = \text{ngram}(x_i)$
    loss: $J = \text{CrossEntropy}(y_i, t_i)$
    BPTT

**func** ngram($x_i$):
    embed: emb = embedding($x_i$)
    hidden1 = linear_layer_1(emb)
    hidden2 = ReLu(hidden1)
    hidden3 = linear_layer_2(hidden2)
    out = log(softmax(hidden3))

### 4.3 LSTM with word embedding

The raw input is a sequence (may not be a complete sentence) in $seq\_len$ dimension and word embedding process shaped input matrices to $(batch\_size, seq\_len, embed\_dim)$. One LSTM cell output the most likely word through reading the history from left to right; then this output was fed in again as new input. Given these, the dimension of $weights$ and $hidden\_size$ in LSTM would be easy to infer. Since the output from one layer should be in the same dimension as the required input dimension of the layer following, we assign $hidden\_size = embed\_dim$ so that the output and input are always in the same dimension. This set is especially convenient when adding multiple layers. Another important hyper parameter is the learning rate. In theory, if the learning rate is too great, the model is likely to skip of the minima of loss and the learning rate is too little will lead to slow convergence. We tried different learning rate to select the locally best one. Considering this, we implement a two layers LSTM here as baseline. Other parameters like bias is initialized by PyTorch package. At each time step, LSTM required a single vector in $embed\_dim$ dimension as $x_t$ , together with hidden state and cell state.

**Pseudocode for LSTM LM**

initialize $hc = (h_0, c_0) = (\mathbf{0}, \mathbf{0})$, where **0** is a matrix $\in \mathbb{R}^{batch\_size \times embed\_dim \times hidden\_size}$
**for** epoch in EPOCHS **do**
    **while** $i < N$ **do**
        prepare input: $\tilde{x}_i = [x_i, \ldots, x_{i+seq\_len}]$
        prepare target: $\tilde{t}_i = [t_{i+1}, \ldots, t_{i+seq\_len+1}]$
        gain predicted word, hidden and cell state: $\hat{y}_i, hc^{new} = \text{LSTM}(\tilde{x}_i, hc)$
        update hidden and cell state: $hc = hc^{new}$
        evaluate error: $err = \text{cross-entropy}(\tilde{t}_i, \hat{y}_i)$
        BPTT
        $i = i + seq\_len$

For supplementary, deeper LSTM was a stack of multiple single LSTM structure. A much deeper LSTM can be constructed by connecting several our models together. We use PyTorch framework to implement LSTM and the setting details were in section 5.2.

# 5 Results

## 5.1 n-gram model

The n-gram model fitted well on a small or medium size training set. We split input text in the form of 80 percent of data for training and 20 percent data for test. The loss decreased through iteration and experiment settings are listed as follows.

| Experiment data | Training set B |
|---|---|
| Epochs | 20 |
| Word embedding dimension | 20 |
| Context size | [2, 6, 11, 15, 20] |

**Table. 1.** Experiment hyper parameters (n-gram)

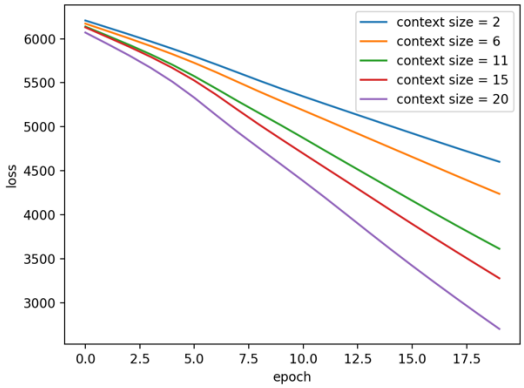


**Fig. 3.** a

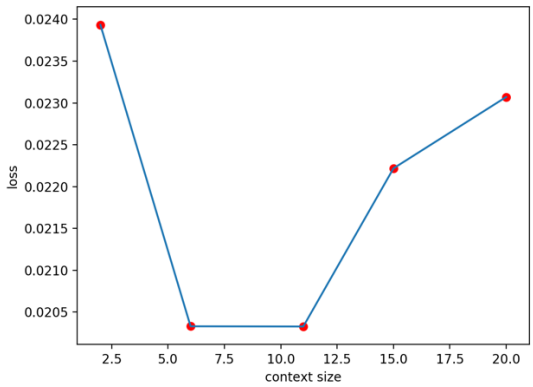


**Fig. 3.** b

Fig.3. a showed the change of loss during training epochs and Fig.3. b showed the loss on test set with respect to each context size in Fig.3. a.

Although training errors on longer context was smaller, the corresponding error on test set arose after a stationary point. N-gram fit highly well on training set because of the exact string matching characteristics. It was suitable for producing output based on seen data set such as similarity check, text categorization and classification. To point out, n-gram hardly works on unseen word combinations. Due to the impractical of long term dependency in n-gram, the predicted word is only reasonable regarding a few words previous but almost not understandable in multiple sentences. Therefore, we did not think the n-gram LM is able to create new articles based on this naive architecture.

## 5.2 LSTM

LSTM learns slowly when set a small $seq\_len$. Given larger dataset and increased $seq\_len$, LSTM performed better than n-gram in the aspect of learning speed and accuracy. When being trained on training set A, LSTM took almost 10 times of epochs in order to result in similar accuracy with n-gram, in the situation with small size vocabulary. Admitting that drawback, we then train LSTM on training set C which is larger. The number of epochs is hard to compare with the one in n-gram because LSTM is tested on a larger dataset than n-gram and still it slide through all the subsequences. If the number of epoch is set the same as n-gram then the model is trained very slow and cause the over fitting problem. Given this, we apply training set C which n-gram model can hardly handle. The results show that LSTM works well. Additionally, we compared the loss from different $seq\_len$ and experiment settings are list as follows.

| Experiment data | Training set C |
|---|---|
| Training iterations | 5000 |
| Word embedding dimension | 50 |
| LSTM input dimension | 50 |
| LSTM hidden size | 50 |
| LSTM layers | 2 layers, with dropout rate 0.5 in the last layer |

| | |
|---|---|
| Fully connected layers | linear layer |
| | ReLu activation function |
| | dropout rate 0.2 |
| | linear layer |

**Table. 2.** Experiment hyper parameters (LSTM)

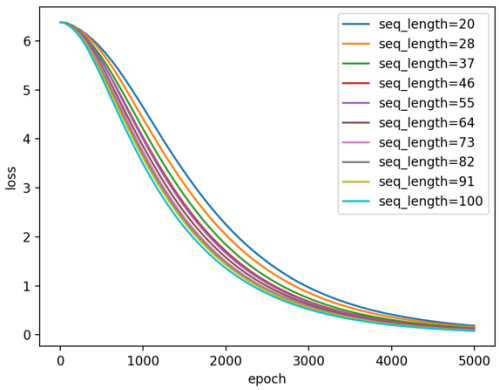


**Fig. 4.** a

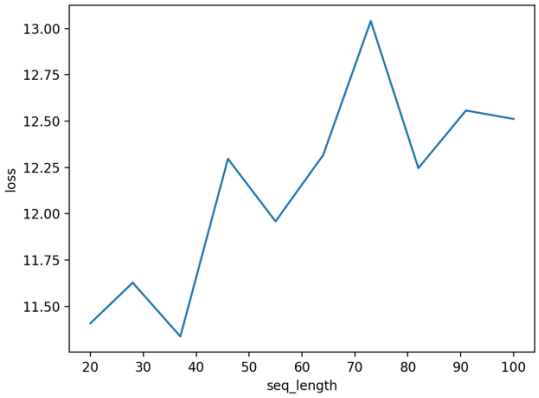


**Fig. 4.** b

Fig.4. a showed the running loss through 5000 epochs; 4b showed the different loss corresponding to different $seq\_length$.

We conclude that input sequence length for one iteration did not affect LSTM model much on training set since they converge at a similar loss value. The reason behind is that the length of a single feed-in has limited influence of the whole LSTM LM. The parameters and states are adjusted through all the dataset and the nature of long dependency will reduce the impact of input length. To be clearer, if the iteration is sufficient for LSTM LM going through the whole dataset several times, then different length of a single sequence result in a similar performance. Meanwhile, the length of dataset in total matters. LSTM will keep predicting one same word or pattern, which has no meaning in language, if the dataset is small. However, as shown in the Fig. 4. b, with increment of the length of sequence, the loss fluctuates strongly. As we mentioned previously, if the iteration is not as many then the input sequence length did require careful selection when modeling on test set. Although the loss of both LM converge to an acceptable amount, LSTM has better extensionality and more robust on unseen words.

## 6 Conclusions and Future Work

We introduced two LM in this paper, n-gram and LSTM, which both are useful in word-level prediction and generating. The pre-processing of data, $\psi(w_{k-l}^{k-1})$, in this paper is word-to-index and word embedding, which is memory efficient and preserve more information of words. For a small size word set, n-gram is fast and agile, with fairly good accuracy. LSTM is more robust and widely used for the ability of long term dependency preservation. However, LSTM learns slowly when the context is short.

As to future work, we would focus on the improvement on LSTM LM. In theory, building larger hidden layer size and deeper neural network architectures could always improve learning. In the aspect of data pre-processing, we thought that the frequency of occurrence of each word is different that some words occur rarely while some words are very likely to occur. Given this, if words whose frequency of occurrence is less than a threshold, we put those in a class naming 'rare class' for example. Because these rare words are hard for LM to learn and make less influence than the common word, we believe this method will improve the efficiency and accuracy of LSTM when modelling language. During the training procedure, split input string as meaningful sentence would result in better predictive performance theoretically. In English, we can set '.' as an ending symbol when preparing inputs so that the learning is based on a whole sentence each iteration.

## References

[1] S. Hochreiter, "Untersuchungen zu dynamischen neuronalen Netzen," Munich, 1991.

[2] S. Katz, "Estimation of probabilities from sparse data for the language model component of a speech recognizer," *IEEE transactions on acoustics, speech, and signal processing,* vol. 35, no. 3, pp. 400-401, 1987.

[3] A. Stolcke, "Bayesian learning of probabilistic language models," 1994.

[4] Y. a. D. R. j. a. V. P. a. J. C. Bengio, "A neural probabilistic language model," *Journal of machine learning research,* vol. 3, no. Feb, pp. 1137-1155, 2003.

[5] H. Schwenk, "Continuous space language models," Spoken Language Processing Group, LIMSI-CNRS, BP 133, 91403 Orsay cedex, France, 2006.

[6] H.-S. Le, I. Oparin, A. Allauzen, J.-L. Gauvain and F. Yvon, "Structured Output Layer neural network language model," IEEE, Prague, Czech Republic, 2011.

[7] M. Sundermeyer, R. Schlu ¨ter and H. Ney, "LSTM Neural Networks for Language Modeling," International Speech Communication Association, Aachen, Germany, 2012.

[8] T. Mikolov, M. Karafia, L. Burget, J. H. Cernocky and S. Khudanpur, "Recurrent neural network based language model," ISCA, Czech Republic, USA, 2010.

[9] M. Sundermeyer, H. Ney and R. Schlüter, "From feedforward to recurrent LSTM neural networks for language modeling," IEEE, Aachen, Germany, 2015.

[10] A. Maiolo, "SR wiki," SR wiki, 10 1 2015. [Online]. Available: http://recognize-speech.com/language-model/n-gram-model/comparison. [Accessed 25 4 2018].

[11] J. Plested, "RECURRENT NEURAL NETWORKS (RNNS)," Canberra, 2018.

[12] I. Sutskever, O. Vinyals and Q. V. Le, "Sequence to Sequence Learning with Neural Networks," NIPS, 2014.