WORD-LEVEL LANGUAGE MODELLING USING AN LSTM NEURAL NETWORK

Julie Maria Petersen (s164510), Lise Styve (s153748)

ABSTRACT

In this paper, word-level language modelling is considered with a simple objective, which is predicting the next word given the previous words within some text. This Natural Language Processing (NLP) task is approached with supervised learning using a Recurrent Neural Network (RNN) with Long Short-Term Memory (LSTM) cells. The model is sought optimized and regularized using weight decay and dropout. A validation perplexity of 88.2 is achieved on the task specific data set Penn Treebank (PTB). When trained on other data sets, the language model enables to adapt to the style and content of the conditioning text, however the generated sequences are not sufficiently realistic and coherent. This can be due to a continued need for regularization and larger amounts of data.

1. INTRODUCTION

Language models have increased in importance as they continuously improve performance in NLP tasks, e.g. by generating coherent and human-like pieces of text. They are used for many purposes such as machine translation, summarization, spelling correction and in text generation which is the area of interest in this paper. RNN's are especially popular for this task and have been used to generate text in several different domains [1].

The objective of this paper was to build a word-level LSTM language model that when trained on textual data was able to generate similar text sequences. This was sought accomplished with inspiration from previous research in the field.

2. LANGUAGE MODELLING

This section presents the different aspects of a neural network with recurrent units (RNN). Further, the methods used in building and training our LSTM language model are introduced.

2.1. Recurrent Neural Network

The reason for the widespread use of RNN's in NLP is that it in contrast to other neural networks allows processing of sequential data. In language modelling, the data needs to be processed in a sequence as each word is interpreted in the context of the previous words. Specifically, the RNN computes the probability distribution $\hat{y}_t$ of the next word x_{t+1}, given a sequence of words $x_1, x_2, ..., x_t$ [2].

$$\hat{y}_t = P(x_{t+1}|x_t, ..., x_1) \tag{1}$$

RNN's support sequential processing by adding a loop to the standard feed forward architectures. This allows stepping through sequential input whilst persisting the state of nodes in the hidden layer between steps [3]. As the input to the RNN consists of words, an embedding layer is added which creates a word vector for each input word and eventually for all words in the vocabulary. Lastly, a dense layer is added to transform the output from the hidden layers to a vector of probabilities. This is done using a softmax function, $softmax(z_t) = \frac{\exp(z_t)}{\sum_j \exp(z_j)}$, that transform the elements of the output vector $\hat{y}_t$ into probabilities. It is the usual choice for multi-class problems like this where each word in the vocabulary is actually equivalent to a class. The overall architecture of this many-to-many RNN is shown in Fig. 1.

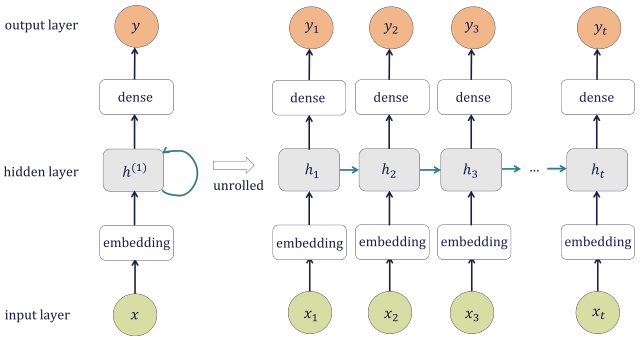

Fig. 1. RNN Language Model

The network is unrolled into a chain of neural networks as seen in Fig. 1, each sharing the same weights and activation functions. After forward-propagation of an input sequence, back-propagation through time (BPTT) calculates loss and error gradients across each of these networks in the chain. Subsequently, the entire network is rolled up and the weights are updated. The performance is thereby optimized by adjusting the weights to reduce loss [2]. This is elaborated in Section 2.4.

However, the standard RNN runs into problems such as the vanishing and exploding gradient. The gradient explodes as a result of repeatedly multiplying gradients that are larger than 1.0 with each other on the way back through the network. The problem is commonly handled using gradient clipping that scales down the gradient when its norm gets above a certain threshold. RNN's struggle to learn Long-Term Dependencies, i.e. to retain information from the beginning of a sequence for longer periods of time. This is caused by the vanishing gradient issue related to back-propagation and gradient descent over multiple time steps. The gradient vanishes as a result of small values being multiplied through the network due to the activation functions used. This problem can however be solved using gated memory cells, popularly LSTM cells [4].

2.2. Long Short-Term Memory

The LSTM neural network is explicitly designed to avoid the Long-Term Dependency problem and can improve performance significantly compared to the standard RNN. The components in an LSTM cell are visualized in Fig. 2.

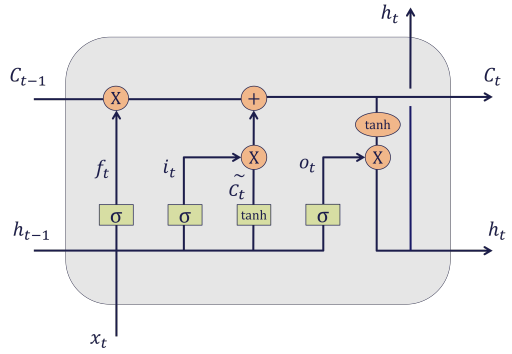

Fig. 2. LSTM Cell

An LSTM cell has the ability to remove or add information to the cell state, carefully regulated by gates that control the flow of information. The mathematical formulation is,

$$
\begin{aligned}
f_t &= \sigma(W_f(h_{t-1}, x_t)) \\
i_t &= \sigma(W_i(h_{t-1}, x_t)) \\
\tilde{C}_t &= \tanh(W_c(h_{t-1}, x_t)) \\
o_t &= \sigma(W_o(h_{t-1}, x_t)) \\
C_t &= f_t \odot C_{t-1} + i_t \odot \tilde{C}_t \\
h_t &= o_t \odot \tanh(C_t)
\end{aligned} \tag{2}
$$

where $[W_f, W_i, W_c, W_o]$ are weight matrices, x_t is the vector input at timestep t, h_t is the current hidden state, C_t is the memory cell state and $\odot$ is element-wise multiplication. The sigmoid function, $\sigma(z) = \frac{1}{1+\exp(-z)}$, pushes the values to be between 0 and 1. This is therefore used when probabilities are needed. This relates to the previously introduced softmax function, however the sigmoid is used when having a two-class problem like this where information should either be forgotten or remembered. The hyperbolic tangent function, $\tanh(z) = \frac{\exp(2z)-1}{\exp(2z)+1}$, pushes the values between -1 and 1 [5].

The first gate in an LSTM cell is a forget gate where irrelevant history is thrown away from the cell state using a sigmoid layer. During BPTT this is what prevents the gradients from vanishing, as the sigmoid layer will always create some values close to 1. Next, the input goes through an update gate that works in two steps. It decides what new information to add to the cell state through a sigmoid layer and creates new candidate values to be added to the state using a $\tanh$ layer. Then the old state is updated to only incorporate the information kept from the forget gate. Finally, the output gate decides what parts of the cell state to output [4].

2.3. Word Embeddings

The approach to representing words in NLP is based on distributional semantics in which a word's meaning is given by the words that it frequently appears close to [2]. In the input layer each word in the vocabulary is mapped to a unique integer. Then the embedding layer provides a spatial mapping of all these words using word vectors. The similarity of these are used to calculate the probability of a word given its context. The model is trained so that an optimal embedding for each word in the vocabulary is learned. This is done by adjusting the word vectors to maximise the probability, so that the difference between input and target word is minimized. This is further elaborated in section 2.4.

Similar words will be closer to each other in the high dimensional embedding space. However, since such a high dimension cannot be visualised, an algorithm named T-distributed Stochastic Neighbor Embedding (t-SNE) is used to reduce the embedding dimensionality. This algorithm converts each embedding vector to 2D such that similar words are nearby points and dissimilar words the opposite [6] [7].

2.4. Optimization

The training of a neural network is posed as a non-convex optimization problem

$$
\min_w \frac{1}{T} \sum_{t=1}^{T} J_t(w) \tag{3}
$$

where T is the size of the training set. To minimize the loss function J_t, different optimization algorithms use different equations to update the weights w of the network [8]. Especially for NLP tasks, there is a tendency to prefer adaptive

learning rate methods such as Adam, which is an extension to the classical Stochastic Gradient Descent. Adam uses estimations of first and second moments of the gradient ΔJ_t to adapt the learning rate for each weight [9].

The loss function of the language model is most commonly the cross-entropy between the predicted probability distribution $\hat{y}_t$ and the true next word $y_t(x_{t+1})$.

$$J_t(w) = CE(y_t, \hat{y}_t) = -\log(\hat{y}_{x_{t+1}}) \tag{4}$$

So the overall loss for the entire training set is

$$J = \frac{1}{T}\sum_{t=1}^{T} J_t(w) = \frac{1}{T}\sum_{t=1}^{T} -\log(\hat{y}_{x_{t+1}}) \tag{5}$$

The exponential of the loss, $\exp(J)$, is called the perplexity relationship. This measure is often used to evaluate the performance of a language model. Low values of the perplexity imply more confidence in predicting the next word in the sequence [2].

2.5. Regularization

Many problems with neural networks are caused by overfitting, wherefore much research has been conducted into avoiding this. The more complex the network, the more prone to overfitting. However, the problem can be helped by implementing different regularization methods and thereby significantly improving the model's performance [8]. Some common methods include L^2 regularization and dropout.

L^2 regularization is a method that adds a regularization term to the loss function. The regularized cross-entropy loss function is

$$J = J_0 + \frac{\lambda}{2T}\sum_{w} w^2 \tag{6}$$

This results in a compromise between finding small weights and minimizing the original loss function J_0. The relative importance of the two is decided by the regularization parameter λ. The method is also referred to as 'weight decay' as it forces the weights to decay towards zero [10] [11].

Another frequently used method is dropout. This literally means 'dropping' some units in the network by randomly setting a chosen proportion of them to zero leaving them non-influential. This introduces more randomness in the model which helps it generalize better. The standard dropout applied on all connections of the network does not work well for RNN's. However, other variations of dropout can reduce overfitting substantially. Instead of applying dropout to all connections, it is only added in the non-recurrent layers, i.e. *not* in hidden-to-hidden relations. The same dropout is used for the inputs and the outputs of the RNN. Dropout can also be applied to the embeddings of the words in the vocabulary. The non-dropped embeddings are all scaled by $\frac{1}{1-p_e}$, where

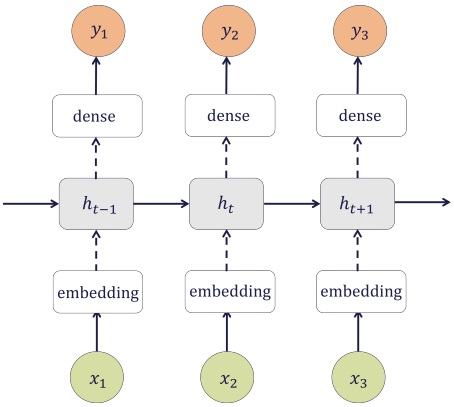

Fig. 3. Dropout is applied in the connections with dotted lines

p_e is the dropout proportion [10] [12]. Fig. 3 shows where the dropout is applied marked with dotted lines.

Other regularization methods can be applied, though the above mentioned are those used in building our LSTM language model.

2.6. Text Generation

There are different methods to choose from when generating text with language models. As mentioned, the language model provides a probability distribution of the next word in the text given the previous words. When generating text, the most obvious way to select the next word is therefore just to choose the word with the highest probability in this distribution. This is referred to as greedy decoding. However, this can result in repetition of a sentence over and over, which surely is not desired. Another approach has therefore been proposed, namely the top-k decoding algorithm. Here the *k* most probable words are randomly chosen between when deciding for the next word. This *k* can be fixed or dynamically adjusted according to the probability distribution currently sampled from. In the last version, the *k* is defined by

$$k = \arg\max_{k} \left[\sum_{i=1}^{k} sort_{descending}(P(x_{i+1}|x \leq i))_i \right] \leq k_p \tag{7}$$

The probability distribution, $\hat{y}_t$, is sorted in descending order, and the highest probabilities are summed over one by one. The *k* is then the number of probabilities used in the sum before it no longer passes being smaller than a fixed threshold $k_p \in [0, 1]$ [13].

3. THE LSTM LANGUAGE MODEL

This section introduces the details, i.e. the specific data and parameters, used in training our LSTM language model. This leads to an evaluation of the model's performance related to different regularization methods as well as word embeddings.

3.1. Details

The approach to building our model has been through research, where inspiration has specifically been taken from three sources by 'Grave, Joulin, Usunier (2016)' [14], 'Zaremba, Sutskever, Vinyals (2015)' [15] and 'Merity, Keskar, Socher (2017)' [8]. The hyperparameters used in the models from these papers have been the starting point for the choice of hyperparameters tested in our model. All three papers build word-level models using the PTB data set. To enable performance comparisons, our model was therefore also initially trained on this data. The PTB data set is widely used within language modelling as it includes text from a wide range of genres. It has a vocabulary of 10.000 words. The training set consists of ∼929k words, the validation ∼73k words and lastly the test set contains ∼82k words.

The hyperparameters used in the models from the research papers and from our best performing model on the PTB data are presented in Table 1 and 2.

Model	Seq.	Layers	Embed.	Hid.
Neural cache LSTM (Grave)	30	1	1024	1024
Non-reg. LSTM (Zaremba)	20	2	200	200
Medium LSTM (Zaremba)	25	2	650	650
AWD-LSTM (Merity)	50	3	400	1150
Our LSTM - all reg.	35	2	650	650

Table 1. Size of input sequence, number of layers, embedding size and hidden size

Model	Batches	Epochs
Neural cache LSTM (Grave)	20	50
Non-reg. LSTM (Zaremba)	20	15
Medium LSTM (Zaremba)	20	40
AWD-LSTM (Merity)	40	750
Our LSTM - all reg.	20	75

Table 2. Number of batches and epochs

From Table 2, it can be seen that the number of epochs varies much across the different models. Initially, we observed the training over a high number of epochs and then stopped the training as the model started to overfit, that is when the validation error started to increase instead of decrease. This is a type of regularization referred to as 'early stopping'.

With regards to the other regularization methods, 0.1 was used for the embedding dropout and 0.5 for the locked dropouts between the non-recurrent layers (ref. Fig. 3). Furthermore, $\lambda = 2 \cdot 10^{-5}$ was used for the weight decay (ref. Eq. 6). All the weight matrices in the LSTM were initialized uniformly from $\mathcal{U}(-\sqrt{\frac{1}{H}}, \sqrt{\frac{1}{H}})$, where $H = 650$ in our model (ref. Eq. 2). The embedding weights were uniformly initialized in the interval $[-0.1, 0.1]$.

For the optimization, the Adam optimizer with a learning rate of 0.001 and the cross entropy loss function (ref. Sec. 2.4) was used. The gradient was prevented from exploding by clipping the norm of the gradients at 5.

3.2. Results

3.2.1. Performance

The validation and test perplexity for different models trained on the PTB data set are presented in Table 3.

Model	Validation	Test
Neural cache LSTM (Grave)	86.9	72.1
Non-reg. LSTM (Zaremba)	120.7	114.5
Medium LSTM (Zaremba)	86.2	82.7
AWD-LSTM (Merity)	60.0	57.3
Our LSTM - all reg.	88.2	84.5
Our LSTM - w/o embedding dropout	93.3	90.7
Our LSTM - w/o non-recurrent dropout	105.4	100.3

Table 3. Validation and test perplexity on the PTB data

From the performance results, it can be seen that our LSTM model with a validation perplexity of 88.2 performs far better than the Non-reg. LSTM (Zaremba) as expected. Our model almost reaches the same perplexity as the Neural cache LSTM (Grave) and Medium LSTM (Zaremba). Adding more regularization would most likely improve the performance of our model in the direction of the AWD-LSTM that incorporates many other regularization methods. It can further be seen from the table that the non-recurrent dropout has the highest impact on the performance.

3.2.2. Learned Word Embeddings

The learned embedding space is visualized using t-SNE for a set of chosen words and those nearest to them. Specifically, the 15 words closest to the words 'that', 'germany', 'consultant' and 'week' are visualized in Fig. 4. Their relation in the full embedding space is shown in Fig. 5.

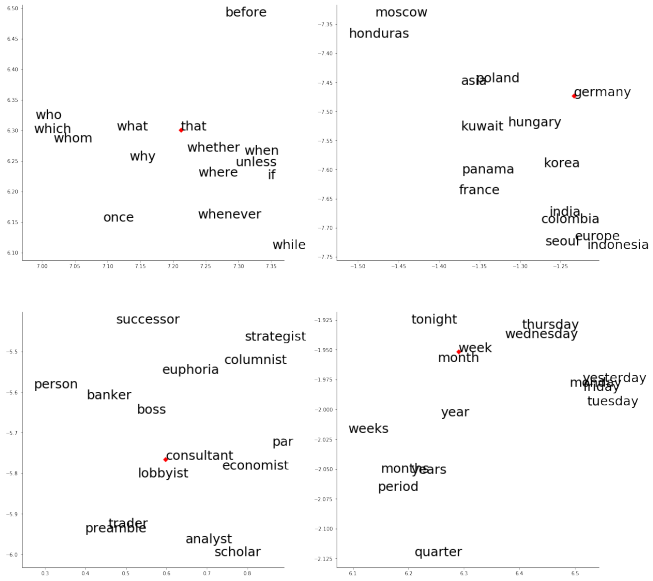

Fig. 4. Word embedding visualization using t-SNE on the 15 most similar words to four selected words

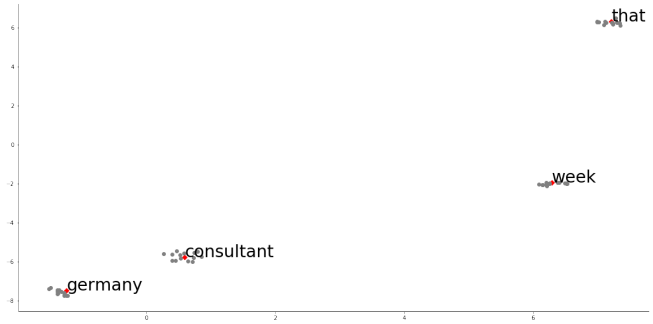

Fig. 5. The learned relations in the full embedding space

Based on the visualizations, the learned word embeddings seem to capture patterns quite well as similar words are in fact placed nearby in space. The word 'germany' is only surrounded by other countries, and 'week' is only surrounded by other terms relating to the calendar. The meaning of these words are therefore successfully represented by their context.

3.3. Experimenting with Text Generation

Our LSTM model's ability to generate text was tested by training it with the same hyperparameters on other data sets. First on storylines from the American animated series Kim Possible and secondly on the first three Harry Potter books by J.K. Rowling.

3.3.1. General Experiment Details

The data sets were split 80:20 in a training set and validation set. Then 'early stopping' was used to decide for the optimal number of epochs for the training. Using the earlier mentioned decoding algorithms (ref. Sec. 2.6), the final models were used to generate text sequences based on the data. The prediction was initiated by two words of our choice. For both data sets the vocabularies included signs, e.g. a comma was considered a word. Further, contractions such as 'don't' were *not* separated. The most known names were kept with capital letter, while all other words were changed to lower case. Words generated after a full stop were changed to start with capital letter.

3.3.2. Kim Possible

The Kim Possible data set had a vocabulary of 6.433 words. The training set contained $\sim$45k words and the validation set $\sim$11k words. This data set is small, which made it quite hard to prevent overfitting, and we already stopped the training at 20 epochs. Some coherent sentences however still occurred in the text generation. We used the fixed top-k algorithm and experimented with different k for the decoding.

Fixed top-k algorithm with $k = 3$:

Rufus says that he didn't mention that he is a meanie and that he is a radio talk show doctor. Kim says that he can get a job at bueno nacho. Kim says that cost is not the issue and that stopping Lucre is a point of pride. Ron thinks that he needs to get a job with the two of them.

Fixed top-k algorithm with $k = 5$:

Kim runs into her. Ron then says he has found the brain switch machine. He says that they need to find an important mission. Kim then tells her mom that she is a sweetheart, to which Ron is forced. As Ron tries to play matchmaker as she is a pair. Ron then points that he has to go to the conclusion. Kim is then greeted out and Ron says that he has been observing the kimmunicator. The two bullies call the kimmunicator.

Some parts of the sequences seem quite coherent, however it is clear that the model does not have the sufficient amount of data to perform optimally.

3.3.3. Harry Potter

Since the success for the small data set with Kim Possible was limited, we further trained the model on text from the Harry Potter books. This had a vocabulary of 16.957 words. The training set contained $\sim$204k words and the validation set $\sim$51k words. The training was stopped at 25 epochs.

Greedy decoding algorithm:

Harry told him. He was looking at the other end of the school. He was looking at the other end of the school. He was looking at the other end of the school. He was looking at Harry and Ron.

Fixed top-k algorithm with $k = 5$:

There was nothing else. Harry looked around his shoulder for a second, it didn't feel like that he might have to leave, he didn't look at it, but at all the other of people was looking at Harry. He couldn't see what he was doing. Harry shook the hand as they went back. A small photograph was shaking his hair as he went in. "what is it?" said Hagrid at the door. He jumped to his feet with a large laugh and Harry had a long, sharp face. He didn't see it in a mirror.

Adjusted top-k algorithm with $k_p = 0.35$:

Ron cried suddenly. They had a sudden route to the fire. Harry and Ron were still looking at each other. "what are you doing?" said Harry. "well, i don't know how i would i see it..." he let out a little sigh and then said, "i'm sorry, Harry, i've got to go to the hospital wing." " but what are you talking about?" said Harry." well, i know i was in the car." "yeah," said Hagrid, " but if i'd got to keep it out of the way." "i know that," said professor Mcgonagall, standing up and piling them up the corridor.

From the examples above, some patterns are apparent. As expected the greedy algorithm gets stuck on a sentence. The coherence seem to be similar for both versions of the top-k algorithm, however the adjusted *k* seems to generate more dialogue than the fixed *k*. Overall, the sequences are though still quite flawed. This is likely due to lack of regularization and data.

4. CONCLUSION

The objective of this paper was to build a word-level language model with LSTM cells. These type of gated cells enable learning Long-Term Dependencies that are hard to capture with a standard RNN. Inspired by three chosen research papers, the model was initially trained on the PTB data set. This enabled performance comparison and further eased the specification of the hyperparameters and regularization methods. For our model, the methods used were weight decay, embedding dropout and dropout between the non-recurrent layers. This resulted in a best performing model with a validation perplexity of 88.2, which was better than one of the models used for comparison. The non-recurrent dropout had the most impact on this improvement since removing it resulted in a validation perplexity of 105.4. The performance could most likely be further improved by implementing even more regularization. The same hyperparameters from the best performing model was finally used to train LSTM language models on the storylines from Kim Possible as well as the first three Harry Potter books. This enabled generation of sequences from these two contexts that did in some cases appear coherent and meaningful, however not flawless demonstrating room for improvement.

5. REFERENCES

[1] A. Karpathy, "The unreasonable effectiveness of recurrent neural networks," 2015.

[2] "Natural language processing with deep learning," CS 224N, Stanford University, 2019.

[3] M. West, "Explaining recurrent neural networks," Bouvet Oslo, 2019.

[4] C. Olah, "Understanding lstm networks," Colah's Blog, 2015.

[5] S. Sharma, "Activation functions in neural networks," 2017.

[6] C. Maklin, "t-sne python example," 2019.

[7] C. Olah, "Deep learning, nlp, and representations," Colah's Blog, 2014.

[8] S. Merity, N.S. Keskar, and R. Socher, "Regularizing and optimizing lstm language models," ArXiv.org, 2017.

[9] D. P. Kingma and J. L. Ba, "Adam: A method for stochastic optimization," 2014.

[10] A. Vidhya, "An overview of regularization techniques in deep learning (with python code)," 2018.

[11] M. A. Nielsen, "Neural networks and deep learning," 2018, Determination Press.

[12] Y. Gal and Z. Ghahramani, "A theoretically grounded application of dropout in recurrent neural networks," 2015.

[13] A. Aghajanyan, "Importance of decoding algorithms in neural text generation," 2019.

[14] E. Grave, A. Joulin, and N. Usunier, "Improving neural language models with a continuous cache," ArXiv.org, 2016.

[15] W. Zaremba, I. Sutskever, and O. Vinyals, "Recurrent neural network regularization," ArXiv.org, 2015.

The data and code used in the implementation is found at `github.com/s164510/Deep-Learning-Project`.