Binarized LSTM Language Model

Introduction.

Neural network models is one of the new and widely used type of models, used for natural language processing (NLP) tasks. They can calculate the probability of the next word, given the short and fixed size context. To improve the performance for long-term dependencies, recurrent neural network models (RNN) are used. Such gate-based structures as long short-term memory (LSTM) (Hochreiterand Schmidhuber,1997) improve the performance of RNN even more, but still have one important drawback: high memory usage for large vocabulary operations due to using multiple embedding layers with word embedding parameters as floating values.

The solution for this is introducing of binarized embedding language model (BELM), where consumption of memory is reduced by representing words as binarized vectors. Also, if in addition we binarize all the parameters of the LSTM language model, we can achieve compression of the parameter space.

Description of the LSTM language model.

Typical language model computes probability of a sentence(x_1, . . . , x_N), where x_t is a word, as:

$$P(x_1, ..., x_N) = \prod_{t=1}^{N} P(x_t | x_1, ..., x_{t-1})$$

LSTM model operates not words, but sequences. A common LSTM unit is composed of a **cell**, an **input gate**, an **output gate** and a **forget gate**. The cell remembers values over arbitrary time intervals and the three *gates* regulate the flow of information into and out of the cell.

For one-layer LSTM network, where N is the length of the sentence, let's denote x_t as the input at the t-th moment, y_t is the output at the t-th moment, which is equal to x_{t+1} in a language model, h_t and c_t as the hidden vector and the cell vector at the t-th moment. Let's initialize h_0 and c_0 with zero. Given x_t, h_{t-1} and c_{t-1}, the model calculates the probability of outputting y_t. In the further formulas, W is the weights of the input connections, where the subscript $_q$ can either be the input gate $_i$, or output gate $_o$, the forget gate $_f$ or the memory cell $_c$, depending on the activation being calculated. As we use vector notations, c_t is not just one cell of one LSTM unit, but contains h LSTM unit's cells.

$$\begin{aligned}
\boldsymbol{f_t} &= \text{sigmoid}\left(\mathbf{W_f}\left\{\boldsymbol{h_{t-1}}, \boldsymbol{e_t}\right\} + \boldsymbol{b_f}\right) \\
\boldsymbol{i_t} &= \text{sigmoid}\left(\mathbf{W_i}\left\{\boldsymbol{h_{t-1}}, \boldsymbol{e_t}\right\} + \boldsymbol{b_i}\right) \\
\boldsymbol{o_t} &= \text{sigmoid}\left(\mathbf{W_o}\left\{\boldsymbol{h_{t-1}}, \boldsymbol{e_t}\right\} + \boldsymbol{b_o}\right) \\
\hat{\boldsymbol{c}}_t &= \tanh\left(\mathbf{W_{\hat{c}}}\left\{\boldsymbol{h_{t-1}}, \boldsymbol{e_t}\right\} + \boldsymbol{b_{\hat{c}}}\right) \\
\boldsymbol{c_t} &= \boldsymbol{f_t} \cdot \boldsymbol{c_{t-1}} + \boldsymbol{i_t} \cdot \hat{\boldsymbol{c}}_t \\
\boldsymbol{h_t} &= \boldsymbol{o_t} \cdot \tanh\left(\boldsymbol{c_t}\right)
\end{aligned}$$

The word probability distribution at the t-th moment can be calculated as:

$$P(y_t | x_1, ..., x_t) = \boldsymbol{p_t} = \text{softmax}(\mathbf{W_y} \boldsymbol{h_t})$$

The probability of taking y_t as the output at the t-th moment is:

$$p_{y_t} = \boldsymbol{p_t} \times \boldsymbol{y_t}$$

Binarized Embedding Language Model

In case if we will binarize input and output embeddings in LSTM, we will get a binarized embedding language model (BELM). Let's denote V as vocabulary size, and H as embedding

and hidden layer size, then we will have the size of input and output embedding equal to 4VH, and size of LSTM cells equal to $32H^2 + 16H$. So total size of the memory, needed for 1-layer LSTM will be $8VH + 32H^2 + 16H$. In case if V much greater than H, the most memory is consumed by input and output embeddings, so as soon as if we will binarize it, input and output layers can take 1/32 of the original memory consumption, so the total amount of memory will go down to $0.25VH + 40H^2 + 16H$.

We cannot binarize W as a simple sign function, as in the back-propagation step of LSTM, the float versions of the embeddings are updated according to the gradient of the binarized embedding and will be turned to zero. A typical weight initialization method initializes each neuron's weights randomly from the Gaussian distribution $N(0,\sqrt{1/H})$. This initialization approach can maximize the gradients and mitigate the vanishing gradients problem. The binarization function will look like following in this case:

$$\text{binarize}(w) = \begin{cases} +\sqrt{1/H} & if\ w > 0, \\ -\sqrt{1/H} & otherwise. \end{cases}$$

The weights will be binarized to a floating point number which will be recorded separately, and matrix will save one bit per neuron.

Since directly binarizing the input embeddings W_e and the output embeddings W_y will limit the scale of the embeddings, additional linear layers (without activation) are added behind the input embedding layer and in front of the output embedding layer to enhance the model. Denote W^b_e and W^b_y as the binarized weights corresponding to W_e and W_y. Denote W_{te} and b_{Te}, W_{ty} and b_{Ty} as the weights and the biases of the first and the second linear layer. The input of the LSTM e_t and the word probability p_t of the binarized embedding language model are:

$$\begin{aligned} \boldsymbol{e_t} &= \mathbf{W}_{\mathbf{T_e}}\left(\mathbf{W}_{\mathbf{e}}^{\mathbf{b}}\boldsymbol{x_t}\right) + \boldsymbol{b}_{\boldsymbol{T_e}} \\ \boldsymbol{p_t} &= \text{softmax}\left(\mathbf{W}_{\mathbf{y}}^{\mathbf{b}}\left(\mathbf{W}_{\mathbf{T_y}}\boldsymbol{h_t} + \boldsymbol{b}_{\boldsymbol{T_y}}\right)\right) \end{aligned}$$

For even more memory space economy, LTSM parameters matrix can be binarized as well.

In a binarized linear layer, there are three parameters: binarized matrix W, vector γ (fixes the scale problem of the binary matrix) and vector b, which is bias of the linear layer. Using it, and input x, we can have the algorithm for propagation of linear layer:

$$\begin{aligned} \mathbf{W^b} &= \text{binarize}(\mathbf{W}) \\ \boldsymbol{s} &= \mathbf{W^b}\boldsymbol{x} \\ \boldsymbol{y} &= \boldsymbol{s} \cdot \exp(\boldsymbol{\gamma}) + \boldsymbol{b} \end{aligned}$$

And back-propagation of linear layer:

$\frac{\partial C}{\partial \boldsymbol{b}} = \frac{\partial C}{\partial \boldsymbol{y}}$

$\frac{\partial C}{\partial \boldsymbol{\gamma}} = \frac{\partial C}{\partial \boldsymbol{y}} \cdot \boldsymbol{s} \cdot \exp(\boldsymbol{\gamma})$

$\frac{\partial C}{\partial \boldsymbol{s}} = \frac{\partial C}{\partial \boldsymbol{y}} \cdot \exp(\boldsymbol{\gamma})$, $\frac{\partial C}{\partial \mathbf{W^b}} = \frac{\partial C}{\partial \boldsymbol{s}}\boldsymbol{x}$, $\frac{\partial C}{\partial \mathbf{W}} = \frac{\partial C}{\partial \mathbf{W^b}}$

$\frac{\partial C}{\partial \boldsymbol{x}} = \frac{\partial C}{\partial \boldsymbol{s}}\mathbf{W^b}$

update $\mathbf{W}$, $\boldsymbol{\gamma}$, $\boldsymbol{b}$ according to $\frac{\partial C}{\partial \mathbf{W}}$, $\frac{\partial C}{\partial \boldsymbol{\gamma}}$, $\frac{\partial C}{\partial \boldsymbol{b}}$ with learning rate η.

$\text{clamp}(\mathbf{W}, -\alpha, \alpha)$

return $\frac{\partial C}{\partial x}$

The embeddings are binarized and additional linear layers are added after the in-put embedding layer and in front of the output embedding layer. However, the additional linear layers are also binarized according to Algorithm1 and Algorithm2.

If to calculate the memory usage of the algorithms above, binarized LSTM (BLLM) will have an advantage towards BELM in LSTM cells size, so the total amount of used memory for it will come down to $0.25VH + 1.25H^2 + 16H$.

The described algorithm was evaluated on several large datasets:

- The Penn TreeBank corpus with a vocabulary size of 10K and 4.8% words out of vocabulary (OOV). It contains 42K sentences with 887K words.
- The Chinese SMS corpus with vocabulary size of 40K, number of sentences 380K and number of words 1931K words

- The SWB-Fisher corpus consisting of 2.5M sentences with 24.9M words and having a vocabulary size of about 30K.

Model test results:

The results of checking the memory consumption for the models differ slightly depending on the corpora for the small number of hidden layers (500): for the PTB and SMS the best results were shown by BELM model, while for SWB corpora – by LSTM model, hence, BELM model results were not too much worse. For bigger size of the hidden layer (1000), BELM model was better in all the cases. BLLM model underperformed in all the cases. It was checked as well using for MEN and WS-353 2 words similarity tasks whether we lose any information representing the embeddings as binary vectors, and for BLLM the results were better than for basic LSTM model, while LSTM outperformed BELM model for WS-353 task.

Conclusion:

In the current review it was described a new language model (BELM), which can solve the problem of occupation large space by NN based language models. As in RNN the memory consumption grows with the size of the hidden layer, it appeared to be effective to represent words (for BELM model) with binarized vectors, which only contain parameters of -1 or 1 and, for further compression, to binarize the parameters of LSTM language model.

References:

1. Xuan Liu ∗ , Di Cao ∗ , Kai Yu, “Binarized LSTM Language Model”, Proceedings of NAACL-HLT 2018, pages 2113–2121. New Orleans, Louisiana, June 1 - 6, 2018
2. Sepp Hochreiter and Jürgen Schmidhuber. 1997. Long short-term memory. Neural computation 9(8):1735–1780
3. Wikipedia, https://en.wikipedia.org/wiki/Long_short-term_memory