Report: Neural Language Model with Multi-layer Perceptron

In this project, we try to implement neural language model with multi-layer perceptron by using 3 words to predict the next word. We have a vocabulary list having 250 words.

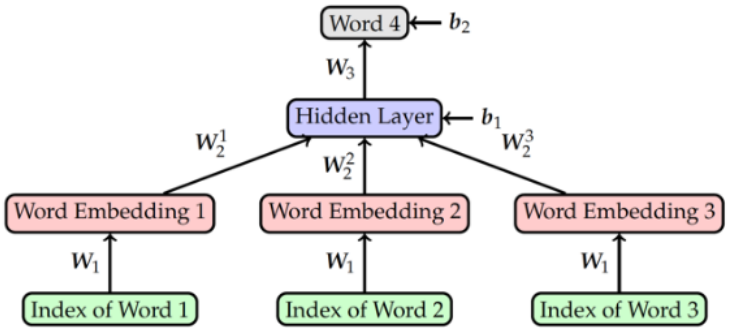

Figure 1: Neural Language Model

In Figure 1, we see that how to apply steps in forward propagation. For the functions and steps, we can follow Neural networks and Deep Learning Book by Nielsen (2015). At first, let define sigmoid and softmax functions and their gradient.

$$\text{Sigmoid function: } f(x) = \frac{1}{1+e^{-x}} \Rightarrow f'(x) = f(x)(1-f(x))$$

$$\text{Softmax function: } g(x)_k = \frac{e^{x_k}}{\sum_{i=1}^{n} e^{x_i}} \Rightarrow g'(x) = \begin{cases} -g(x)_k g(x)_j & if\ i \neq j \\ g(x)_k(1-g(x)_j) & if\ i = j \end{cases}$$

In Figure 1, W_1's are 250x16 dimensional, W_2's are 16x128 dimensional and W_3 is 128x250 dimensional. However, we can applying steps by concatenating W_1's and also W_2's. Therefore, we get 250x48 dimensional weight,let say A_1, as concatenation of W_1's and 48x128 dimensional weight,let say A_2, as concatenation of W_2's. We have data as X having rows which are sum of one hot vectors of 3 words and targets as y which is prediction of the next word. For example, our train data have 372500 and each row has 3 words. That is, it is 372500x3 dimensional. After one hot encoding and summation, we get 372500x250 dimensional training data. Let say $X \in \mathbb{R}^{nx250}$ and $y \in \mathbb{R}^n$. If we use one hot targets, then it will be $\mathbb{R}^{nx250}$ Also, we have bias terms $b_1 \in \mathbb{R}^{1x128}$ and $b_2 \in \mathbb{R}^{1x250}$. Now, we are ready to write **forward propagation** steps:

$$z_1 = XA_1 \rightarrow z_1 \textit{ is } \text{nx48 } \textit{dimensional}.$$

$$z_2 = z_1A_2 + b_1 \rightarrow z_2 \textit{ is } \text{nx128 } \textit{dimensional}.$$

$$z_3 = f(z_2) \rightarrow z_3 \textit{ is } \text{nx128 } \textit{dimensional}.$$

$$z_4 = z_3W_3 + b_2 \rightarrow z_4 \textit{ is } \text{nx250 } \textit{dimensional}.$$

$$\textit{output} = g(z_4) \rightarrow \textit{output is } \text{nx250 } \textit{dimensional}.$$

Also, let define our loss function for this model:

$$\text{Cross-Entropy Loss Fuction: } L(y_i, p_i) = -\sum_{i=1}^{n} y_i log p_i$$

where y_i's are true labels and p_i's are predictions of the model after softmax function. We can also follow Rafay Khan's blog to write steps for the gradients. According to forward propagation steps, we can write **backward propagation** steps:

$$d_{output} = \frac{\partial L}{\partial output} = \frac{1}{n}\frac{y}{output} \Rightarrow \text{nx250 } dimensional.$$

$$d_{z_4} = \frac{\partial L}{\partial z_4} = \frac{\partial L}{\partial output}\frac{\partial output}{\partial z_4} = d_{output} g'(z_4) \Rightarrow nx250 \text{ } dimensional.$$

$$d_{z_3} = \frac{\partial L}{\partial z_3} = \frac{\partial L}{\partial output}\frac{\partial output}{\partial z_4}\frac{\partial z_4}{\partial z_3} = \frac{1}{n} d_{z_4} W_3 \Rightarrow \text{nx128 } dimensional.$$

$$d_{w_3} = \frac{\partial L}{\partial z_3} = \frac{\partial L}{\partial output}\frac{\partial output}{\partial z_4}\frac{\partial z_4}{\partial w_3} = \frac{1}{n} d_{z_4} z_3 \Rightarrow \text{128x250 } dimensional.$$

$$d_{b_2} = \frac{\partial L}{\partial z_3} = \frac{\partial L}{\partial output}\frac{\partial output}{\partial z_4}\frac{\partial z_4}{\partial b_2} \leftarrow \text{mean of} d_{z_4} \text{according to axis 1.} \Rightarrow \text{1x250 } dimensional.$$

$$d_{z_2} = \frac{\partial L}{\partial z_3} = \frac{\partial L}{\partial output}\frac{\partial output}{\partial z_4}\frac{\partial z_4}{\partial z_3}\frac{\partial z_3}{\partial z_2} = d_{z_3} f'(z_2) \Rightarrow \text{nx128 } dimensional.$$

$$d_{z_1} = \frac{\partial L}{\partial z_3} = \frac{\partial L}{\partial output}\frac{\partial output}{\partial z_4}\frac{\partial z_4}{\partial z_3}\frac{\partial z_3}{\partial z_2}\frac{\partial z_2}{\partial z_1} = \frac{1}{n} d_{z_2} A_2 \Rightarrow \text{nx48 } dimensional.$$

$$d_{A_2} = \frac{\partial L}{\partial z_3} = \frac{\partial L}{\partial output}\frac{\partial output}{\partial z_4}\frac{\partial z_4}{\partial z_3}\frac{\partial z_3}{\partial z_2}\frac{\partial z_2}{\partial A_2} = \frac{1}{n} d_{z_2} z_1 \Rightarrow \text{48x128 } dimensional.$$

$$d_{b_1} = \frac{\partial L}{\partial z_3} = \frac{\partial L}{\partial output}\frac{\partial output}{\partial z_4}\frac{\partial z_4}{\partial z_3}\frac{\partial z_3}{\partial z_2}\frac{\partial z_2}{\partial b_1} \leftarrow \text{mean of} d_{z_2} \text{according to axis 1.} \Rightarrow \text{1x128 } dimensional.$$

$$d_{A_1} = \frac{\partial L}{\partial z_3} = \frac{\partial L}{\partial output}\frac{\partial output}{\partial z_4}\frac{\partial z_4}{\partial z_3}\frac{\partial z_3}{\partial z_2}\frac{\partial z_2}{\partial z_1}\frac{\partial z_1}{\partial A_1} = \frac{1}{n} d_{z_1} X \Rightarrow \text{250x48 } dimensional.$$

Therefore, we can write update parameters for A_1, A_2, W_3, b_1 and b_2. Let say the update parameters $A_1^*, A_2^*, W_3^*, b_1^*$ and b_2^* respectively. Also, we need a learning rate, η.

$$A_1^* = A_1 - \eta d_{A_1} \Rightarrow \text{250x48 } dimensional.$$

$$A_2^* = A_2 - \eta d_{A_2} \Rightarrow \text{48x128 } dimensional.$$

$$W_3^* = W_3 - \eta d_{W_3} \Rightarrow \text{128x250 } dimensional.$$

$$b_1^* = b_1 - \eta d_{b_1} \Rightarrow \text{1x128 } dimensional.$$

$$b_2^* = b_2 - \eta d_{b_2} \Rightarrow \text{1x250 } dimensional.$$

While implementing the model with train data (inputs and targets) and validation data, number of epochs is 60 and learning rate is decay for each epoch. Also, mini-batch size is selected as 100. To choose the best of them, I try many different numbers. During working the "for" loops for epochs and mini-batches, there were some problems. At first, I initialize weights as very small numbers between 0 and 1 and biases as 0. When I tried to different learning rates, train loss increased and also validation loss is 0 throughout epochs. Therefore, I get these best hyper-parameters for this model.

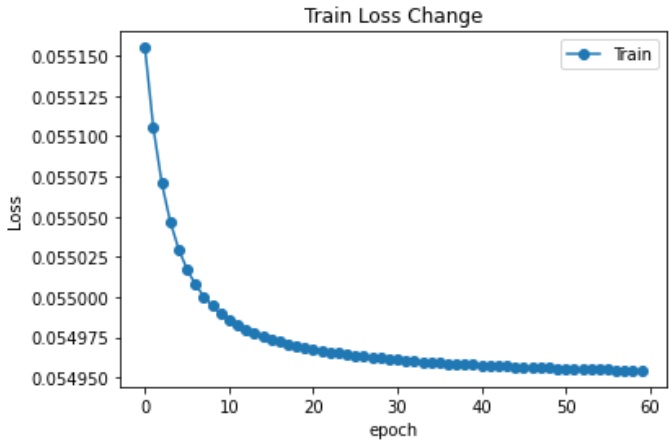

Figure 2: Train Loss Change for 60 epochs

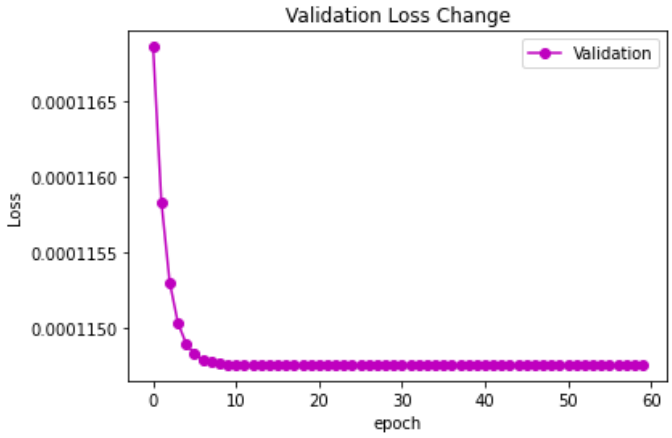

Figure 3: Validation Loss Change for 60 epochs

As above figures, we can see the loss changes for both train and validation data. The train losses are calculated as a mean of losses in mini-batches. As in Figure 2, train loss decreases over time and also validation loss for each epoch is decay as in Figure 3. For validation loss, there are small changes (or decrease) after 10 epochs. On the other hand, the train loss starts to decrease less after 50 epochs. At the end, they decrease as expected. During the training, learning rates are used by decreasing. In Takase et al. (2018) says decay learning rates enables us to better model performance. Therefore, the formula for decay learning rates which decrease by 0.5 is provided as

$$learning\ rate = 0.1 * 0.5^{epoch}$$

Also, we can see that, training accuracy increase along epochs. We can see the change of train accuracy in Figure 4. However, there is a problem in validation accuracy. It is constant and has 0.17380645161290323 value for all epochs;that is, it does not change. Although changing all hyper-parameters or changing initial weights for the model, they did not work. I have controlled many times whether I have made any mistakes in the model, forward propagation and backward propagation steps. I couldn't solve this problem no matter what I tried differently. In the paper of Poggio et al. (2017), overfitting causes this problem. It says choosing smaller learning rates is a solution or decay learning rates for epochs or mini-bathes. I tried them, but again this problem continued.

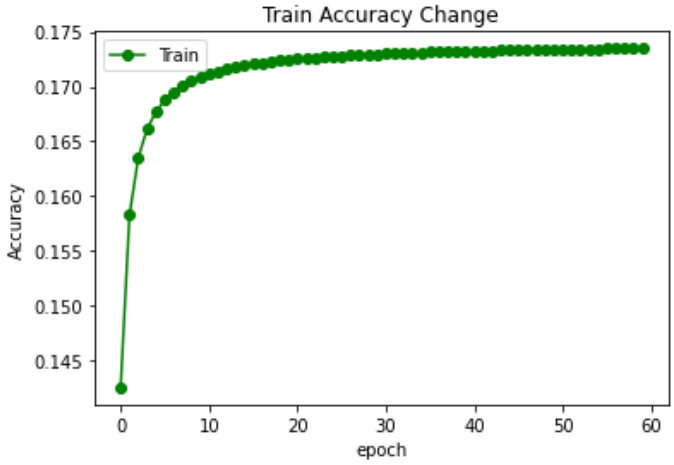

Figure 4: Train Accuracy Change for 60 epochs

For the evaluation of the model, test loss is calculated as 0.00011856766675922448 and test accuracy is 0.1729247311827957. The loss is very small. Thus, we can interpret the result as our model works well. Of course, if we have got better accuracy value, our model would have learned more and we would have achieved better results. However, in general, the decrease in the losses and the increase in the train accuracy throughout the epoch tells us that the model is good but not enough.

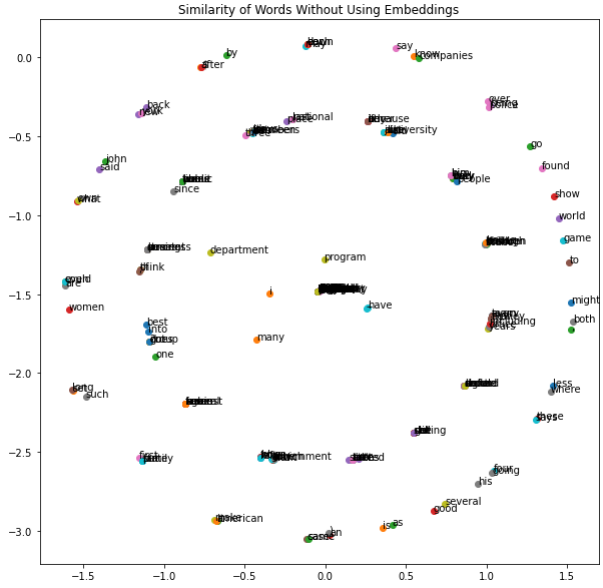

Figure 5: Word Similarity Before Using Word Embeddings

In the Figure 5, we can see the similarity between word before 250x16 word embeddings. Here, we use only the one hot words in vocabulary list and we apply tsne function to them. Actually, in the begining, we cannot say anything about their similarity because the shape looks like a spiral. We expect them to be meaningfully closer to each other. Our word embeddings should have 250x16 dimensions. At the begining, I use concatenation of W_1's as A_1 is 250x48 dimensional. To get word embeddings, we have 3 different W_1's here. So, we have 3 different word embeddings. I use its first 16 columns, columns between 16 and 32 and last 16 columns as W_1's.

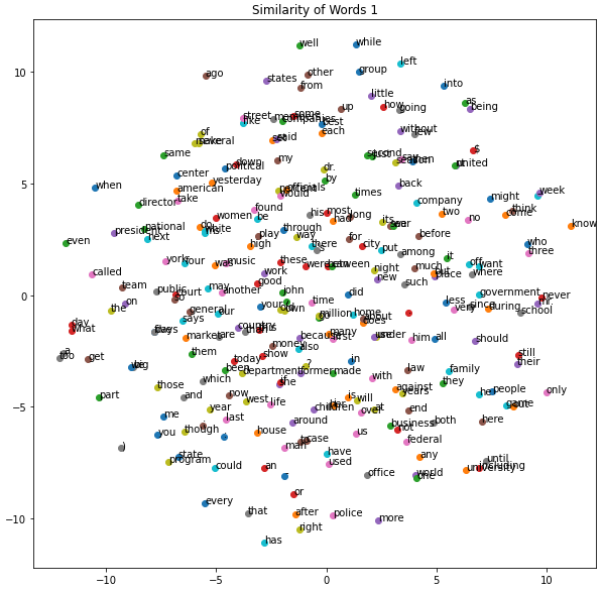

Figure 6: Word Relation After Using Word Embeddings with First 16 Columns

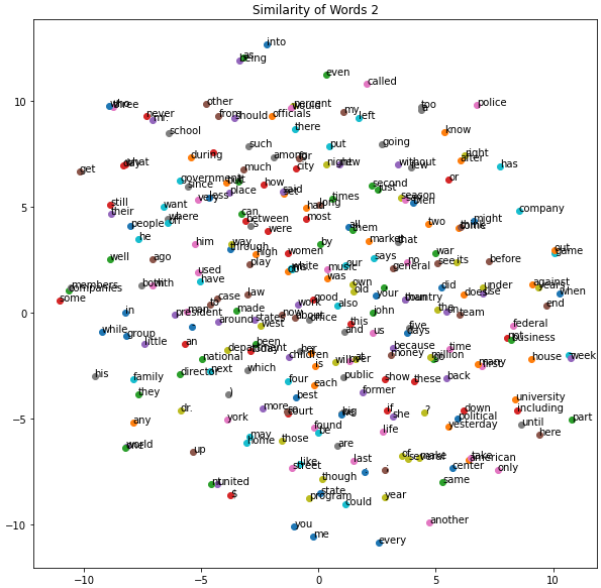

Figure 7: Word Relation After Using Word Embeddings with Columns Between 16 and 32.

As we in the Figure 6, Figure 7 and Figure 8, different W_1's give different word embeddings and therefore, we get different word groups. t-SNE function helps us to work high dimensional data and we obtain insight of the data in lower dimension which is 2 for our case. t-SNE is used to maximize the similarity between two probability distributions which have higher dimension and the lower dimensional space by using embeddings (Van der Maaten et al., 2008). By trainig t-SNE fonction, we get 250x2 dimensional data and we can plot them as in the figures. After the trainig, it gives information about Kullbach-Leibler divergence (KL divergence). According to results,KL divergence after 1000 iterations is 0.389315 for Figure 5, KL divergence after 1000 iterations is 1.388791 for Figure 6, KL divergence after 1000 iterations is 1.402606 for Figure 7 and finally KL divergence after 1000 iterations: 1.402931 for Figure 8. Bigi (2003) says KL divergence is a distance measure and it can be text categorization.

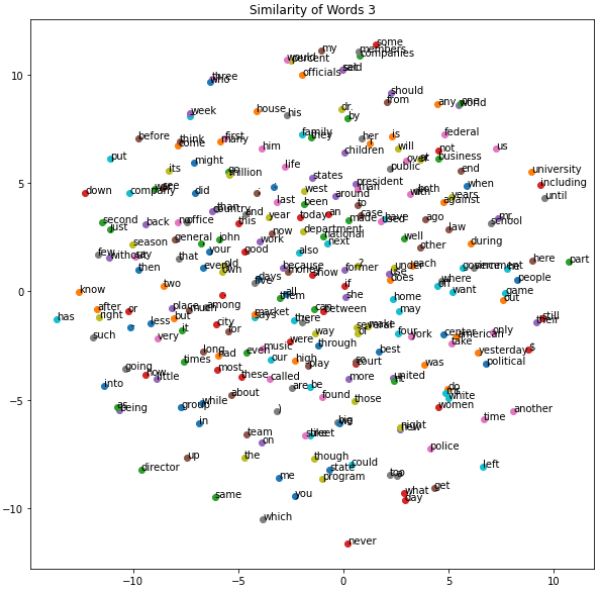

Figure 8: Word Relation After Using Word Embeddings with Last 16 Columns

After getting the results, we can interpret as higher KL divergence value gives better relation between words because KL divergence value before using word embeddings is smaller. Therefore, we will analyze Figure 8. **For a few cluster of words**, we can give an example in Figure 9. For example, 'last' and 'year' are related words and also 'good' and 'work'. We use it side by side in our sentences many times since they are related.

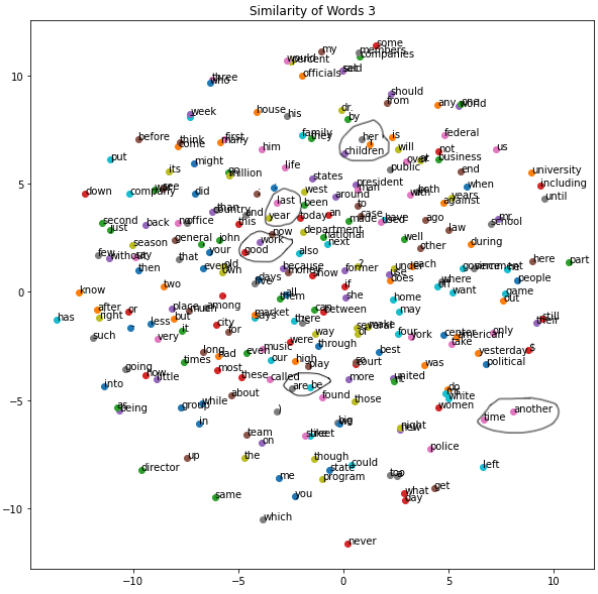

Figure 9: Cluster of Related Words

For the predictions of next word, we get 'city of new .', 'life in the .' and 'he is the .' The predictions are not successful, so our model is not very good at this. However, when we look at first row of train data which is 'going to be', we see that the prediction of the next word is also '.'. So, the predictions cannot be meaningful due to real predicted targets. Of course, the model has some problem since for 3 different input, the prediction is the same.

Reference

- Bigi, B. (2003). Using Kullback-Leibler distance for text categorization. *In European conference on information retrieval*, 305-319. Springer, Berlin, Heidelberg.
- Nielsen, M. A. (2015). *Neural networks and deep learning, 25.* San Francisco, CA: Determination press.
- Poggio, T., Kawaguchi, K., Liao, Q., Miranda, B., Rosasco, L., Boix, X., ... & Mhaskar, H. (2017). Theory of deep learning III: explaining the non-overfitting puzzle. *arXiv preprint arXiv:1801.00173.*
- Takase, T., Oyama, S., & Kurihara, M. (2018). Effective neural network training with adaptive learning rate based on training loss. *Neural Networks, 101*, 68-78.
- Van der Maaten, L., & Hinton, G. (2008). Visualizing data using t-SNE. Journal of machine learning research, 9(11).
- https://d2l.ai/chapter_ multilayer-perceptrons/backprop.html
- https://cmdlinetips.com/2019/07/dimensionality-reduction-with-tsne/
- Rafay Khan. https://www.kdnuggets.com/2019/08/numpy-neural-networks-computational-graphs.html