Training a customizable language model for affect text generation

Francis Ikpe Ogar Damachi
Lab of Human-computer Interaction, EPFL Lausanne, Switzerland

***Abstract*—This paper explains the process when implementing the baseline and affect language model. We are interested in training an affect language model that can generate sentences in a specific affect category. These sentences should be grammatically and semantically correct. Furthermore, we want to see if the affect language model achieves lower perplexity scores compared to the baseline model. For that reason, we train both models and compare their scores. When training the affect language model, we use the LIWC api to generate a vector in order to represent if a word is emotionally colorful or not. In fact, this api is crucial in training the affect LM.**

I. Introduction

Language modelling is an essential part of Natural Language Processing (NLP) which involves predicting the probability of the next word after having observed the previous words[**?**]. This technique can be used for various tasks such as correcting spelling errors, word completion, bot response generation etc. Specifically the use case of a language model that we are interesting in, is not only basic text generation but a model which is capable of generating emotional text based on an affect category proposed by a user. The word affect is defined in the dictionary as an 'expression of emotion or feelings displayed to others through facial expressions, hand gestures, voice tone, and other emotional signs such as laughter or tears'[**?**].

In order to generate emotional text in this project, we will follow the same steps as were accomplished in the proposed solution illustrated in the research paper: **Affect-LM: A Neural Language Model for Customizable Affective Text Generation** by *Sayan Ghosh1, Mathieu Chollet1, Eugene Laksana1, Louis-Philippe Morency2* and *Stefan Scherer1*. We will focus on training both a baseline language model and an affect language model in order to compare the effect of the affect property in a language model. Furthermore, the affect language model will be trained to generate text of 5 affect categories. These are: ANXIETY, POSITIVE, NEGATIVE, ANGER, and SADNESS. Although we closely follow the research paper, we go into much detail in what was actually done.

This paper will first explain the choice of the data-set and the various pre-processing steps that we will carry out in the DATA section. Afterwards, it will thoroughly explain the baseline and affect language model with the details of how it we implement it in the 'MODEL AND METHODS' section. In the RESULTS section, the paper will display and comment on the results achieved by both language models. Eventually, the DISCUSSION section will go through an array of obstacles encountered during its implementation and also how some of them were overcame. This will greatly help the reader to pay attention to several intricacies during the training of language models. Finally the paper will finish up with a conclusion summarizing the project as well as giving possibilities of future work that can be done.

II. Data

One of hardest things in machine learning isn't necessarily the implementation of the algorithm but rather possessing and choosing a reasonable data-set for the task. We train the language model in 2 steps: for the first part of the training process, the Cornell data-set is used in training the baseline and affect language model. Later, the IMDB reviews data-set is used for fine tuning the affect-language model.

A. Cornell Data-set

This data-set is composed of 220,579 movie conversations exchanges between a pair of movie characters. Furthermore, it contains 304,713 utterances. The reason for choosing this data-set is due to its richness. Moreover, the data-set has over 30'000 words which suggests that this large vocabulary size implies that the data-set is quite diverse. Another aspect it offers is the amount of emotional colored words it possesses. For each affect category (anxiety, positive, negative, sadness, anger,) the data-set possesses a substantial amount of words that capture it. This can be shown in figure 1

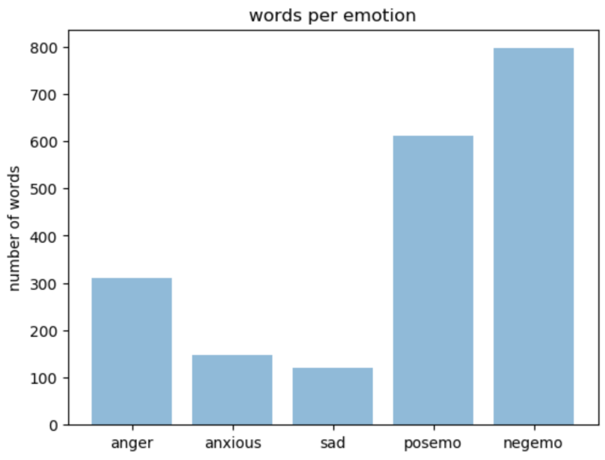


Fig. 1. Plot displaying the number of distinct words in each affect category

B. IMDB reviews

The IMDB reviews data-set, retrieved through Kaggle platform is used for fine tuning. This data-set is considerably less rich and smaller compared to the Cornell data-set. It is composed of two categories of user reviews of movies which are positive and negative reviews. In addition, each category has a range of review scores: positive reviews range from 6-10 while negative reviews are from 0-4. If a score in the positive category is high, the more we hope that the certain review will posses emotionally colorful words capturing the category. We only focus on the positive reviews for the second part of the training process. The decision is purely by preference. Also we choose the IMDB data-set because when a user is satisfied/unsatisfied with something, it is normal for the person to use emotionally colored words to emphasize their satisfaction. Finally, there could be no better place to acquire this other than in user reviews.

C. LIWC

The LIWC is an API which we use to identify if a word/sentence belongs to an affect category. Given a sentence, the API is used to generate a vector of five dimensions, where each dimension has value of either 0 or 1. This vector simply states the presence of the affect category in the sentence. For example, the sentence : '*I am sad while you are happy*' yields a vector:
{ anx : 0, pos : 1, neg : 1, sad : 1, anger : 0 }. As one can see the API does not take into account the syntax nor the semantics of the sentence. It simply analyzes word per word without context and finally checks if this certain word has an affect connotation. More details will be explained of how we build this vector using this API.

D. Google-news

We use a pre-trained word embedding to initialize the weights in the embedding layer of both the baseline and affect language model. The Google-news word embedding is extremely useful because it was trained on 3 billion running words.[**?**] Therefore, the semantics of a given word is well grasped in the vector.

E. Pre-processing

Before feeding the phrases to the model, there are several pre-processing steps we have to effectuate illustrated in Figure 2.

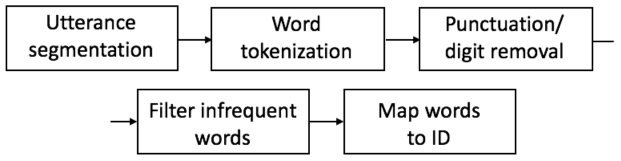


Fig. 2. Schema showing the text pre-processing steps

First of all, we perform utterance segmentation which will make each utterance be a data point. In order to process the words later, we apply word tokenization to each utterance. However, we don't apply basic lemmatization, stop word removal, nor stemming even though these operations will greatly reduce our vocabulary size. This is due to the fact that it defeats the purpose of text generation. In fact these operations will mostly make our sentences syntactically and semantically incorrect. Moreover stemming reduces the variability of our vocabulary which leads to our sentences becoming less rich. Since the training time of the language model will greatly depend on the vocabulary size, we have to try to reduce it as much as possible. We do this by filtering out words that do not occur more than 5 times in the whole corpus. The value 5 as a threshold appears to be a sweet spot which effectively filters words that don't occur quite often and also it does not destroy the meaning of the sentences. Moreover it greatly solves the huge vocabulary size due to spelling mistakes occurring in the corpus. In fact, this operation makes our vocabulary size go from over 30000 to just 15720. Finally, we transform the words into unique indices.

III. Model and Methods

Before training the affect language model and to see if it works correctly, we first have to train the basic Language model or the baseline.

A. Baseline Language model

In order to train the baseline model, we use the definition of n-grams[**?**] This is shown in this equation 1.

$$P(w_1^n) = P(w_1)P(w_2 \mid w_1)P(w_3 \mid w_1^2)..P(w_n \mid w_1^{n-1})$$

$$= \prod_{i=1}^{n} P(w_k \mid w_1^{k-1})$$

Where (w_1^n) is a sequence of n words. In this equation, we try to calculate the probability of a certain sequence. Basically this value is calculated by conditioning each word with the previously already observed words. In code, each term in the product is estimated by applying the $softmax$ function shown in equation 2.

$$P(w_t = i \mid w_1^{t-1}) = \frac{exp(W_i^T f(w_1^{t-1}) + b_i)}{\sum_{i=1}^{V} exp(W_i^T f(w_1^{t-1}) + b_i)}$$

$f(w_i^{t-1})$ is the output of the LSTM network which takes the words i to $t-1$. t simply corresponds to the position where the word appears in the utterance. This is then multiplied with a weight matrix W. The b_i represents a bias term. Finally V stands for the vocabulary size. Vectorizing this equation yields a vector b and a matrix W having these dimensions :

$$W = \Re^{V \times h}$$

$$b = \Re^{1 \times V}$$

$$f(w_i^{t-1}) = \Re^{h \times 1}$$

$$h : number of hidden neurons$$

We make use of a sub-type of a Recurrent Neural Network (RNN) called Long short term memory (LSTM). The reason we use LSTM rather than the basic RNN is because the RNN suffers from the vanishing gradient problem[**?**]

The vanishing gradient problem is a phenomena which occurs during back-propagation. When the error is derived with respect to parameters of the weight matrix, these quantities tend to get smaller as we traverse backwards in the network. This results in the event in which the model is unable to learn long term dependencies as we go closer to the input.[**?**] The inability to learn long term dependencies defeats the purpose of a recurrent neural network. Fortunately an LSTM solves the issue.[**?**]

1) Implementation Details: We train the basic language model using KERAS tensor flow. Luckily, this library provides all the functions necessary. In KERAS, one can easily stack network layers together when building the architecture of the network. Therefore, the first thing we must do is to apply the pre-processing steps as explained in DATA section. We add an embedding layer to the architecture which will be initialized with the pre-trained weights of the Google news word embedding. Of course we have to make sure the dimensions of the embedding layer be the same as the Google news word embedding. Initializing the weights with Google news word embedding and training the word embedding when training the language model will help have more more specialized embeddings for our corpus. We then stack the embedding layer with x LSTM layers. Each LSTM cell will have h_1 hidden neurons. Also we keep the number of time-steps to be variable due to the utterances having variable lengths. For that reason, we don't pad the sequences with zeroes. After, we take the output of the LSTM layer to be the input of a Dense KERAS layer. The Dense layer is a basic neural network with 1 hidden layer that contains n neurons. We let the reader try out different values for the number of hidden neurons. Finally we insert the output of the Dense layer into the $softmax$ function shown in equation 2. An illustration in figure 3 will help clarify the architecture of the baseline model.

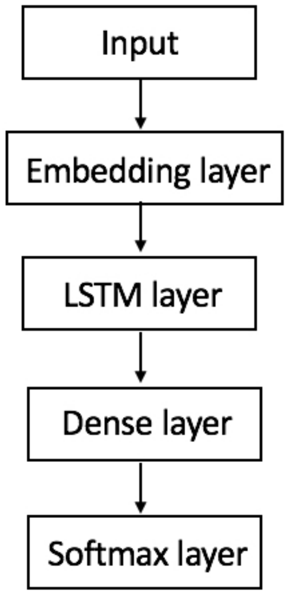


Fig. 3. schema of the baseline model

B. Affect Language Model

As stated in the introduction, we don't really deviate from the idea used in research paper. In order to represent affect in the language model, we replace equation 2 by this one.

$$P(w_t = i \mid w_1^{t-1}, e_1^{t-1})$$

$$=$$

$$\frac{exp(W_i^T f(w_1^{t-1}) + \beta E_i^T g(e_1^{t-1}) + b_i)}{\sum_{i=1}^{V} exp(W_i^T f(w_1^{t-1}) + \beta E_i^T g(e_1^{t-1}) + b_i)}$$

The W_i^T corresponds to the weights of the Dense layer in figure 3 and $f(w_1^{t-1})$ is once again the output of the LSTM layer. What fundamentally changes is the addition of an energy term β which is a constant but bounded between 0-5 for our case. It determines how emotionally colorful the type of text we want to generate. As β tends grow larger, the more emotionally colorful the text the model will learn. Notice that when β is zero equation 3 is exactly the same as the baseline equation 2 meaning the absence of affect.

The $g(e_1^{t-1})$ is the output of a single hidden layer neural network. This perceptron takes as input e_1^{t-1} which is a vector corresponding to the emotional content of the text of length $t-1$. E_i^T is a vector which is represented by a dense layer in KERAS. Finally b_i is once again the bias term. Vectorizing this equation yields vector and matrix sizes having these dimensions :

- $W = \Re^{V \times h}$
- $E = \Re^{V \times h_1}$
- $b = \Re^{V \times 1}$
- $e = \Re^{5 \times 1}$
- $f = \Re^{h \times 1}$
- $g = \Re^{h_1 \times 1}$

h_1 : the number of hidden neurons in the emotional neural network part.
h : the number of hidden neurons in the basic neural network part.

1) Implementation details: The pre-processing of the text is done similarly to the baseline model. However we have to build the emotional vector. We do this using the LIWC API. For each word in our vocabulary, the LIWC gives the value of the affect term we are interested in. Therefore, we extract the affect categories such as anxiety, positive emotion, negative emotion, sadness, and anger to generate a vector of truth or false values. A true value signifies the presence of the affect term whereas a false value means its absence. Eventually, we build the emotional vector by combining the OR logic operator with all the vectors of the words in the utterance. Finally we transform the Boolean values to yield the final vector to 0 and 1.

We need to feed the data in mini-batches since our data-set is quite large when training. This means the training parameters will be updated after seeing a subset of the data-set rather than the whole data-set. For that reason, a batch generator will generate batches of utterances combined with the emotional counter parts. Please refer to the code to see how we implement this technically.

Since we have to generate a prediction at each time-step, we make use of the KERAS Timedistributed layer. This special layer makes it possible to replicate the product $E_i^T g(e_1^{t-1})$ across all time-steps. Once again a detailed implementation is available in this link[**?**] The affect language model structure is illustrated in figure 4.

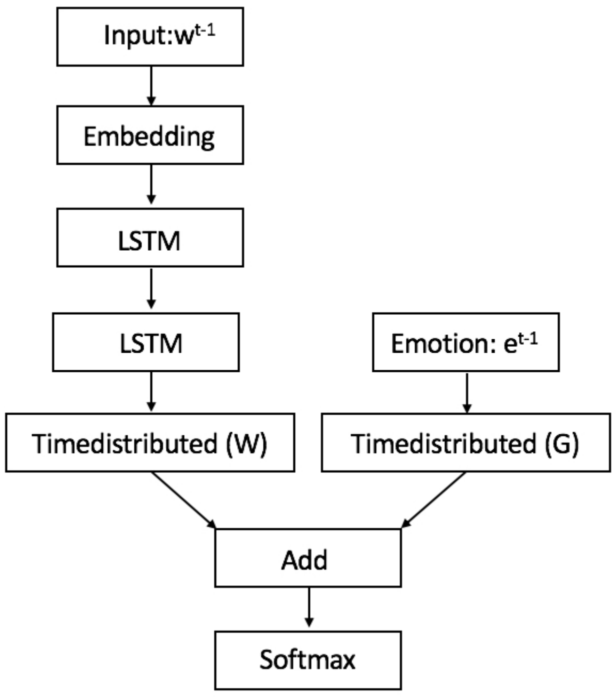


Fig. 4. Schema showing the text pre-processing steps

IV. Results

A. Training parameters setting

1) Baseline: These are the parameters we set for training the baseline architecture network.

- Epochs = 40
- # of neurons in the Dense layer = 200
- # of hidden units in LSTM cell = 300
- # of LSTM layers = 2
- Activation of Dense layer = $softmax$
- Optimizer = $Rmsprop$

2) Affect language model:

- Epochs = 40
- # of neurons in the Dense layer = 200
- # of hidden units in LSTM cell = 300
- # of LSTM layers = 2
- Activation of Dense layer = $softmax$
- Activation of perceptron g = $sigmoid$
- Optimizer = $Adam$
- β: 1,2,3,4

Both baseline and affect language use the categorical cross entropy as the loss function, also they use the perplexity as a measure. In KERAS, the perplexity measurement isn't implemented, so we apply this equation 3 to calculate at each iteration the perplexity.

$$2^{-\sum_x p(x) log_2(p(x))}$$

For the baseline and affect language model, we split our data into training, validation and test set based on this ratio. $0.75 : 0.15 : 0.10$.
Table 1 shows the scores on the training and validation perplexities we achieved for the baseline and the affect language model. We can see that the Cornell data-set achieved a lower perplexity for both and the affect Language model. This is most likely due to the fact that we were training on a bigger data-set. Which suggest that a bigger data-set had more diversity. In fact the vocabulary size is larger than the IMDB reviews. In addition, if we measure the difference between both techniques used to train the language, the affect language model produced on average lower perplexity measurements.

We also visualized the word embeddings of the matrix weights U and V in equation 2. We used the tool in Tensorboard projector to plot the word embeddings in a 3D space using Principle Component Analysis (PCA).

The word embedding U appeared to capture the meaning of a word. Words of similar meaning tended to be clustered together. We noticed that the model tried learning an embedding V which captured the emotion of the word. For example in figure 6, words that where in the same affect category tended to be clustered together. The blue labels represent the positive emotions whereas the red labels represent the angry words. We could see that there was a segregation between these 2 categories. Although it is far from being perfect, as we selected neighbouring words next to a certain emotion,we saw similar emotions in its neighbourhood.

We finally reach the moment where we wanted to generate text. The way it was done was that we sampled a text beginning from the data-set for instance : *i feel*. Then we applied the pre-processing steps that were explained previously. Afterwards, we freely chose the affect type of sentence we wished to generate by creating the emotion vector. Finally we fed the 2 vectors into the trained model. The prediction of the model generated a vector of size V. Each value of the vector is the probability of the word with index i of being

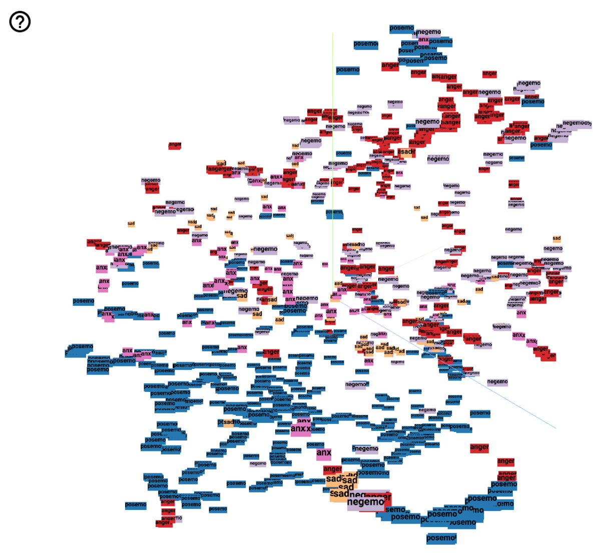
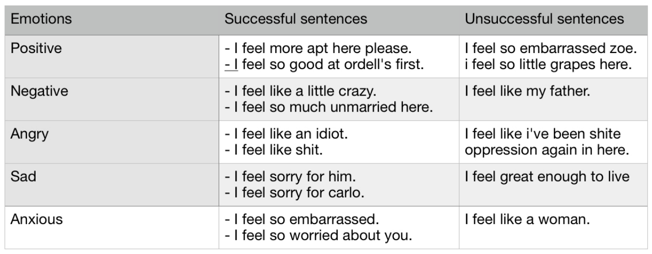


Fig. 5. Schema showing the learned emotion word embedding V

selected as the next word. However, we didn't choose the most probable word directly as the next word. The reason is that if the most probable word wasn't good enough, we didn't want the model to be restricted on the bad choice it had made. For that reason, we sampled from a distribution where most of the time the word with the highest probability would get chosen but sometimes, the model was allowed to explore other possibilities. In that way the model could be more creative. We generated a few sentences per affect category for a given sentence beginning. This is illustrated in Table 2.



Emotions	Successful sentences	Unsuccessful sentences
Positive	- I feel more apt here please. - I feel so good at ordell's first.	I feel so embarrassed zoe. i feel so little grapes here.
Negative	- I feel like a little crazy. - I feel so much unmarried here.	I feel like my father.
Angry	- I feel like an idiot. - I feel like shit.	I feel like i've been shite oppression again in here.
Sad	- I feel sorry for him. - I feel sorry for carlo.	I feel great enough to live
Anxious	- I feel so embarrassed. - I feel so worried about you.	I feel like a woman.

Fig. 6. Table showing the generated senteces

As we can see, the model was able to generate grammatical correct sentences. Also some sentences corresponded to the correct affect category. Unfortunately for some sentences, despite being grammatically correct, they weren't able to capture the affect category. Even though we changed the sentence beginnings, it wasn't able to produce sentences of the correct affect category. This suggested that the model was far from being perfect and major improvements needed to be made.

Perplexity scores of models trained on both data-sets		
Dataset	Baseline	Affect LM
Cornell	64.23	55.35
IMDB reviews	69.27	67.04

V. Discussion

A. Grasping concepts

Before starting to code both language models, it required a detailed understanding of how vanilla recurrent neural worked and its mathematical concepts such as backpropagation through time (bptt). Understanding at first these concepts, made the whole LSTM structure easier to grasp. Also it further reinforced why vanilla recurrent neural network wasn't used instead but rather the LSTM.

B. Debugging

There is a famous saying that writing code takes 10% of the time whereas 90% of time will be spent on debugging. This was exactly what happened.

1) Wrong input: One of the first mistakes was feeding the model sequences of the same lengths. Meaning that we combined all the utterances into one large text. Afterwards, we gave the model as input phrases of length 20 by splitting it to equal lengths of 20. This was quite problematic because the model would most likely not see normal sentence beginnings. Afterwards when we wanted to generate text, upon sampling a sentence beginning from the original data-set such as *The*, the model had probably never seen a sentence beginning with this specific word, so it would generate quite gibberish text. To overcome this minor bug, we proceeded in feeding the model utterances instead to solve the nonsensical text generation.

2) Padding sequences: As explained in the model section, we trained the model on batches of utterances. Since the utterances had different sizes, we had to make the utterances have the same size within a batch. In fact, in KERAS Tensorflow, the library doesn't know how to deal with batches of different lengths. At first, we proceeded in 0 padding the utterances so that they may have the same lengths within a batch. Unfortunately, this solution wasn't really good because, the zero padding vectors were participating in the calculation of the loss function. The effect of this zero padding was that the loss and accuracy during training had outstanding values, whereas the model would generate extremely short sequences of max length 3. This meant that the model was considering the zero padding as a legitimate input and it was favoring its generation. Instead, we changed our approach and didn't specify a fixed number of unrolled time-steps for the model. We grouped the utterances of the same lengths into bins. For example, a bin k contains utterances of length k. Afterwards when we were about to generate a batch, we generated a random number between $[0, k]$. That number would be determine which bin we retrieve the batches. This solution was one of the turning points in the debugging phase. First of all, we were able to train in batches which sped the training process. Secondly the zeroes weren't participating in the loss function, since we omitted the padding. Finally the model started to generate sentences with longer and meaningful text.

3) Optimizer: The choice of the activation played a key role in the training process. This helped yield lower perplexity scores. At first we were using $RMSProp$ optimizer which is an optimizer normally used in training Language models. [?] Due to time constraints, we did not go into detail in understanding the way it worked. Simply in KERAS, we just had to specify it as a key word. The result of using this optimizer was that at the beginning of each epoch, the loss and perplexity would always increase by a factor of 3. Then it will gradually decrease during the epoch. However the decrease wasn't too substantial, meaning that difference between the loss and perplexity between 2 epochs wasn't too significantly different. To solve this we switched to the $ADAM$ optimizer. This drastically changed the training process of the language models, that is,the loss and perplexity decreased between epochs. The model was learning better thanks to the $ADAM$ optimizer.

C. Choice of Preprocessing

Data pre-processing was the one of the most time consuming tasks of the project. It involved thorough data exploration, which would decide the words we would filter from our data-set. For example, we had to decide how we wanted to remove the punctuation. In the corpus, we noticed that there were many words that were built with the dash(-) symbol such as *fruit-loop, hedge-pig, prison-movie, full-on*. These are known as compound words. We had to decide how to treat these kinds of words by either replacing the dash(-) symbol by a space which would yield 2 separate words or keep the dash symbol to make a compound word, hence increasing our vocabulary size. Another problem we noticed was that an utterance like: *No more. Or I call another lawyer. This is the biggest case of your life. Don't try to negotiate. Thirty percent. Say yes or no.* which had full stop marks within the utterance was quite problematic. Removing the full stops from this utterance for example, would yield a sentence which wasn't too meaningful. Fortunately, there weren't too many utterances that were like this, therefore we proceeded in removing them. We simple took into account that the model might observe some utterances which weren't too syntactically correct.

D. Training time

Since the data-set and the vocabulary size was quite big, it greatly affected the number of trainable parameters. In fact the model had over 17 million parameters to train. So we needed to train the model with a GPU. Luckily for us, Google colab offered free GPU and CPU time. This helped a lot because we could write code in a Jupyter notebook within Google colab. It made writing code quite interactive, hence it helped alleviate the pains of debugging. For example, we didn't need to run the complete code to figure out that a matrix had a certain dimension, rather we could simply run the desired code portion. Unfortunately Google colab didn't provide unlimited computing time. In addition their CPU and GPU wasn't really as powerful as expected. This suggested that we needed to use a much more powerful CPU and GPU for the training. Thanks to the efforts of our supervisor:(Dr Pearl PU) and the teaching assistant (Mr. Yubo Xie), we were able to rent CPU and GPU instances from the IC EPFL cluster. This helped speed up the training process by a factor of 1.5.

Despite having these powerful tools, it was still difficult to maximize the computation power of the GPU's through parallel programming. Parallelism means splitting the data into independent portions and making each thread work on each separate data-set portions. However, in an LSTM architecture, the weights are shared in the entire architecture. So it makes it quite difficult to parallelize. This does not mean that it cannot be achieved. But it required a lot of time, work and understanding of parallel programming.

E. Alternate implementation

This method as shown in figure 6 did not contain the Timedistributed layer rather we basically represented all n-grams of a given utterance. Instead of feeding the model this utterance *I don't feel good* for example and predicting at each time-step using the Timedistributed layer, we fed to the model:

[I,I don't, I don't feel, I don't feel good]

with targets as

[don't, feel, good, eos]

This meant that we predicted only 1 word for each n-gram. Although this way of representing the language model corresponded to equation 2, the time we took to train the whole data increased exponentially, since we were increasing the size of the data-set with the n-gram generation. Therefore this solution wasn't quite feasible.

VI. Future work/Conclusion

Finally we conclude this research paper with a quick summary of the steps that were taken in training the affect language. We explain and illustrate the pre-processing steps presented in the Data set section. In the models section we explain in detail the baseline and the affect language model. We also show the techniques with an intuitive schema. Later we display the results that were achieved by showing the perplexity scores of the baseline and the affect LM, the learned emotional embedding V and some generated sentences by the model where some of them weren't too successful in capturing the affect category. In the discussion section, we discuss the obstacles encountered and also the pains during the project. These are the time spent in debugging and training the model. Even though the alternative solution remains mathematically correct but unfeasible, we explain it to give the reader more insight. Finally we explain the choices one has to take when applying some NLP pre-processing techniques.

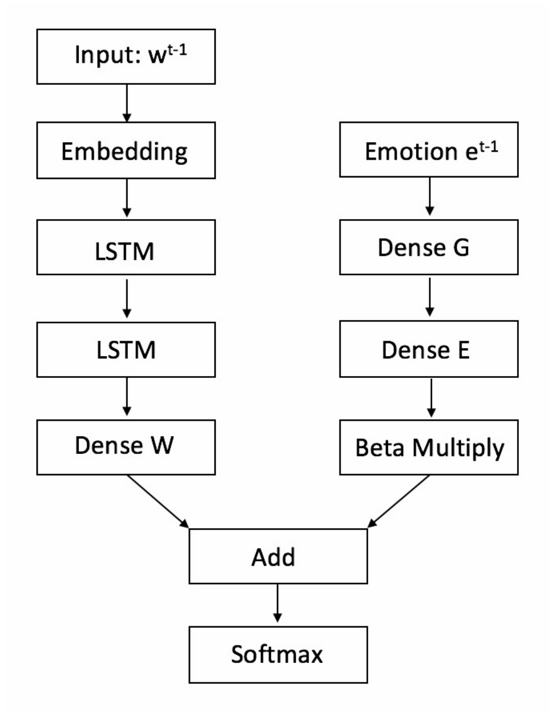


Fig. 7. schema of the baseline model

1) Future work: As shown, the generated sentences are far from perfect. For that reason, we can try to train the model with larger betas to see if the model will eventually start capture bettering the emotional category. Moreover, perhaps having a more powerful GPU will speed up of the training process. What can also be done to debug further the model is to train the model to generate only 2 sentiments which are positive or negative. Another approach is, we can train the model on another data-set that is as large and as diverse as the Cornell data-set and compare the generated sentences.

Acknowledgements

I would like to thank my supervisor Dr. Pearl Pu and the teaching assistant Mr.Yubo Xie for having the time to help me overcome various obstacles encountered during the project. In fact, this project would not have been able to be done without their conscious efforts. It required hard work and perseverance.