Tweet Sentiment Analysis using a Convolutional Neural Network

Team 'MSV'
Hugo MOREAU (238523), Robin SOLIGNAC (235020), Axel VANDEBROUCK (236377)
Communication Systems, EPFL Lausanne, Switzerland

***Abstract*—This paper explores the use of a convolutional neural network (CNN) to estimate the likelihood of a given tweet to express a positive or negative feeling. This network is used on top of pre-trained word vectors. It shows the power of such representation combined with a CNN. Moreover it shows that the fine-tuning of hyperparameters of this network yields moderate performance improvements due to the excellent feature representation offered by the trained vectors. Performances of the model are compared against a simple logistic regression using the same word representation in order to isolate the power of the CNN itself.**

I. Introduction

With the rapid growth of online data, text classification has become one of the key technique to analyse data automatically. We could think of some applications : doing online polling for future election by extracting the sentiment of tweets concerning a candidate, or getting feedback from customers upon the release or presentation of a new product. These classifications are to be realised on larger and larger sets of data, and are therefore at the heart of research for more efficient models. This project aims to do text classification at a sentence level. More specifically to analyse the sentiments enclosed in tweets (binary classification : positive or negative). To do so, a large data set of 2 270 482 distinct tweets is used as a training set (due to computation time, a smaller subset of 181 321 distinct tweets is also considered). The final goal being to predict on 10 000 tweets. Text classification usually relies on features developed by humans, one of the strength of this model relies in the lack of need for any human developed features. However this project uses a clean data set where sentences have already been tokenized/cleaned and a few modifications would be necessary to apply it to a different set of data.

II. Methodology

Neural network are today's most promising and powerful machine learning techniques in general. After some research, it appears that most of 'state of the art' sentence classification algorithms use neural networks. So we decided to propose an efficient solution to the problem using such technology, in particular CNN.

A. CNN for sentiment Analysis

Convolutional neural networks have useful properties that are easy to understand in the context of Computer Vision. Location invariance and local compositionality make intuitive sense for images : **location invariance** (should you be trying to identify an object in an image, the location of this object in the image should yield no importance.) and **local compositionality** (each filter creates a local patch of low level features into a higher level representation, e.g. building edges from pixels, shapes from edges etc.).

However, we loss much intuitions in the case of natural language processing (NLP). For such tasks, each word is represented by a vector (those can be pre-trained like GloVe or *word2vec* or generated from scratch). Then a sentence is represented by a matrix which rows are the vector representation of the words that compose it. This matrix is the equivalent of our image. In vision, filters slide over different patches of the image whereas in NLP, those filters typically have the same width as our matrix (i.e. the length of a word vector representation) and the height depends on how many words we want to slide over (typically 2 to 7 words). The nice intuitions about the usefulness of the CNN's properties for Computer Vision are now much more difficult to understand in the case of NLP. Pixels in a local patch are likely to be related while it isn't always true for words. Usually words in a sentence can be separated by several other words without changing the deep meaning of the sentence (e.g. "This woman is beautiful" could also be expressed by "The woman by the church over there, with the red dress, is so beautiful"). Thus it is hard to see how the compositional aspect plays a role, but also what does higher level representation actually means ?

So it isn't so obvious that CNN are a good match for NLP and maybe we would rather use a more sequential model (reading a sentence from left to right being more common). But it turns out that CNN are actually performing quite well on those tasks. One of the top arguments supporting the CNN are how **efficient** they are. First of all they are fast, convolutions are a central part of computer graphics, and most modern computers are equipped with GPUs to speed up this kind of operations. Moreover they are efficient to represent n-grams as these are embedded in the filters. Computing all the 4-grams can be quite expensive for a large vocabulary (even Google doesn't provide more than 5-grams representations).

B. Composition of the convolutional neural network used

Our neural network is a standard version of a CNN applied to text classification with one layer dedicated to each of the 3 important steps (which will be explained later on). The implementation (in Python using Tensorflow, under Apache licences) originates from Denny Britz based on an article of Yoon Kim [1] on tasks similar to ours.

The first layer is the **embedding layer**. It translates each word into its representation in a vector of chosen length k (300 in Google's w2v, 128 in our implementation). Hence the input matrix is a $n \times k$ matrix representation of the sentences (tweets). Where n is the maximum number of words in a sentence.

The **second layer applies (convolution) filters** on the embedding matrix. Each convolution produces tensors of different shapes depending on the filter height. A sublayer is created for each of those filters. Those layers are finally merged into a big 'feature' layer. Each filter is represented by its weight matrix W and bias b. Then if we let h be the height of a filter (here h is the number of words over which the filter is applied) then we create a feature f_i as :

$$f_i = g(W \cdot x_{i:(i+h-1)} + b)$$

where g is a non linear activation function. We used the Rectified Linear Unit (ReLu) in order to avoid the vanishing gradient problem.

This filter is applied to all windows of words $\{x_{1:h}, ..., x_{n-h+1:n}\}$ to obtain a feature map $f = [f_1, ..., f_{n-h+1}]$ (one for each filter). Finally max-pooling is applied to each of those maps to extract the most meaningful feature for this filter (in Fig. 1 it can be observed that more than 1 filter per filter size is used, namely T). This is how from those multiple filters we can extract the most meaningful features. Doing this for each filter we combine the results into one long feature vector of length $T \times F$ where F represents the number of different filter sizes.

Finally the **last layer is a 'conventional' NN final layer**, which combines the features obtained by the previously explained operation using another weight matrix and bias term. It is composed with a dropout protocol to avoid overfitting. As seen in class, dropout will randomly disable some nodes to prevent them from co-adapting to the presence of other nodes and learning in a more individual manner. The fraction of the nodes that we drop is an hyperparameter that needs tuning (which will be discussed later on). This layer will output scores that are then be used to generate predictions using matrix multiplication and to finally choose the class (negative or positive feeling) yielding the highest score.

C. Data and preprocessing

The provided dataset of tweet had already been tokenized correctly and preprocessed : urls had been deleted and replaced by <url>, the user names by <user> (We also noticed that capitalization had been removed, this is quite unfortunate as they could have been used to detect specific emotions).

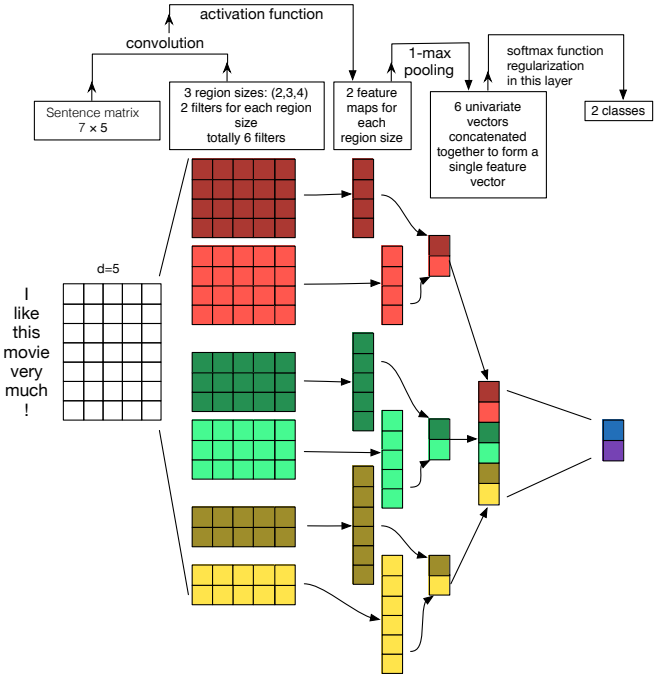

Figure 1. Graphical representation of the net

The only remaining common preprocessing possibility is to delete all non-alphanumerical characters. But these characters are almost surely containing information about sentiment (e.g. emojis, hashtags, ...) therefore we decided to keep them. Hence no preprocessing was done on the tweets themselves. But we surprisingly noted that the training sets contained quite a lot of duplicated tweets ($\approx 10\%$). Those were removed as we didn't want to over-train the network on those specific tweets.

To compute the **embedding matrix**, three different approaches were considered :

1) Random vectors for each word, with arbitrarily chosen dimensions
2) Precomputed word2vec vectors from Google, made from 300 billion words from Google news (those had dimensions $k = 300$ [1])
3) Vectors trained using unsupervised learning with an arbitrarily chosen dimension of $k = 128$ (using gensim word2vec library [2])

From those three methods, we considered only the last one. First of all it was the one yielding the best results and secondly it fortunately was less computationally expensive as it would reduce the size of the embedding matrix compare to the google model.

[1] https://code.google.com/archive/p/word2vec/
[2] https://radimrehurek.com/gensim/models/word2vec.html

D. CNN Hyperparameter

From a theoretical point of view this CNN has 4 hyper-parameters: the shape of filter in the convolution layer, the number of such filters per filter size, the dropout rate and the L2-regularizer. Only the two first will be properly tested as we state in section II-F that the two latter don't have much impact on performance.

To select them we run a grid search over a set of possible choices (can be observed in Table II) that have been chosen according to values used in similar model [1], [2] and by considering that our sentences are always short (less than 140 characters)

We should select the best pair by doing a k-fold cross validation on the training set for each possible label. However such a process would take too much time to compute on personal computers. So we just run one step of a 10-fold (train on arbitrary 90% and test on the remaining 10%) CV on only the small tweet data set. Finally for each pair we use the same seed. So that all pairs are trained on the same 90% of data.

E. Optimization : loss and accuracy

From the scores obtained after the final layer, we can define our loss function. This loss must represent the amount of error made by our network. The standard loss function for classification is the cross entropy[3]. Though most of the times, choices were based on prediction accuracy that is a much more 'human-understandable' quantity. Also since models are trained on batches (see section II-G), and in order to be able to compare the loss across different batch sizes and input sets, the loss mean is recorded rather than the sums.

F. Regularization

The two popular methods used to enforce regularization in a neural network and avoid overfiting are L2-regularisation and dropout. Recent studies [2] indicate that in the case of a CNN with a low number of layers neither of them has an important impact on the system. They will in the best-case scenario increase the accuracy by 1.0% .

Empirically, tests on the implementation confirm these affirmations. Thus, neither of those two is used. Also, due to time constraints computationally expensive tools are avoided. No dropout allows a faster convergence to optimal parameters, also not considering L2-regularization enables to avoid cross validating an optimal value.

G. Training our network

As already discussed, the network needs to learn the weights and bias used for each filter and also in the final layer.

First of all, given the amount of data we had at our disposal, we used stochastic gradient descent over mini-batches to work on the network. Those batches are randomly selected across the input data and are created as many batches as needed to realize all the necessary epochs (number of time we wish to pass over the full training set).

To construct this network we use TensorFlow which allows to work with a neural network much more easily as it will figure out by itself which variables can be "trained" and will compute the gradients automatically to optimize the network. In our implementation we use the `AdamOptimizer`. This first order gradient based optimizer was also used in [1].

We then note that the batch size is another parameter that needs to be considered. With infinite time, the size of batch would not impact the result of the algorithm which will converge to the same value. A large size increases the time to perform a step, but will yield better results for a given number of steps. Empirically, it seems to be the case that for the same amount of time it is better to compute more gradient descent steps than to have a more efficient gradient but less step. These observations match the ones of other studies [2], [1]. So we'll use here a quite small batch-size of 64.

III. Results

This section will discuss the results yielded by the different approaches that are considered in our model.

A. Baseline model

To be able to compare our model to some baseline we use a simple regression where the tweets are represented using the average of the word vectors as computed using our vocabulary (refer to II-C). This allows to evaluate the performance of the network itself isolating the word representation.

To evaluate this baseline model we use a cross validation with 10 folds on the full training set and compute the accuracy [4]. This allows to also see how regularization helps the model to avoid over-fitting on the large training set which could then be compared with the effect of regularization on our neural network. The logistic regression with fine tuned parameter will yield a maximum accuracy of 70.37%.

B. Word vector representations

Table I presents the effect of different strategies for the word representations on the accuracy. We present the results from Kaggle when training on the small set with filter shapes (3,4,5) and 128 filters per such shape. But those results hold under the optimal parameters (presented at III-D).

We can see that our own w2v resulted in better performances and also allowed us to reduce the complexity of computation. Also, even with random vectors, we see that the neural networks can learn useful features and get higher

[3]https://en.wikipedia.org/wiki/Cross_entropy

[4]the Cross validation can be observed in `baseline.ipynb`

Method	Score
Random Vector	0.80520
Google w2v	0.82000
Our own w2v	0.83020

Table I
COMPARISON OF THE RESULTS OBTAINED WITH THE DIFFERENT DATA PREPROCESSING METHODS

performance than with the baseline model on top of a 'smart' representation.

C. Hyperparameters : Filter sizes and number of filters

In table II, we compare different settings of filter shapes and number of filters per shapes. These accuracies are computed using a batch size of 64, with one epoch, no dropout or L2-regularization on the small training set.

Accuracy	64	128	200	300	400	500
2	0.8183	0.8213	0.8230	0.8193	0.8192	0.8224
3	0.8223	0.8222	0.8193	0.8203	0.8233	0.8200
4	0.8216	0.8224	0.8209	0.8213	0.8216	0.8239
5	0.8224	0.8217	0.8213	0.8205	0.8226	0.8187
6	0.8216	0.8191	0.8210	0.8240	0.8231	0.8208
(2,3,4)	0.8215	0.8225	0.8216	0.8168	0.8192	0.8197
(3,4,5)	0.8228	0.8255	0.8165	0.809	0.8212	0.8192
(4,5,6)	0.8230	0.8223	0.8191	0.8193	0.8186	0.8195
(2,3,4,5)	0.8229	**0.8259**	0.8159	0.8201	0.8201	0.8233
(3,4,5,6)	0.8226	0.8214	0.8140	0.8204	0.8197	0.8201

Table II
COMPARISON OF THE ACCURACY FOR DIFFERENT SHAPES (LEFT) AND NUMBER OF FILTERS PER SUCH SHAPE (UP)

This grid search identifies filter shapes (2,3,4,5) along with a filter size of 128 filters per shapes as the optimal parameters.

D. Final result

From all preliminary results, we use the following parameters which seem to be optimal: No dropout nor L2 regularization was considered. Filters of shapes (2,3,4,5) and with 128 filters per shape (on the full training set, with mini-batch of 64 and 1 epoch). Moreover we used our own pre-trained w2v model. The training of the model takes 6 h 59 min and yields a F1-score of 0.8680 on the test set (obtained from Kaggle).

IV. DISCUSSION

Given the very large amount of training data, one of the most challenging part of this project was to tune the hyper-parameters. Training the net took a very large amount of time and the options were limited to employ a systematic testing of our parameters. Even a single CV step on the small data set takes more than 20 minutes on a recent computer[5]. Those times could have been greatly improved by a factor of 3 to 5 on a machine with GPU support of TensorFlow.

However, we found some very helpful articles to guide us toward good initial guesses on the value of such parameters. One of the most useful literature [2] analysed the effect of multiple hyperparameters on various data sets. This allowed us to grid search through values which we knew made sense and not to have to run through too much combinations of those. In fact the number of possible configurations even for our simple model is large : input word vector representations; filter region size(s); the number of feature maps; the activation function(s); the pooling strategy; and regularization terms (dropout/l2).

Hence not all the possible configurations were tested, as mentioned previously we tried only a very small amount of them, and we've not been able to test of all them in an optimal way.

Also, one could try to have multiple channels for the word representation, using the static word representation from our w2v and also some random vectors allowing task specific feature learning on those.

We would also like to mention that some other models could have been tested to achieve better accuracies : especially recurrent convolutional neural networks [3]

V. CONCLUSION

Hence we have seen that by itself word representations play a very important role in sentence classification as it can be interpreted as feature engineering for the sentences. Then combined to the power of a neural network we obtain a very satisfying score. However it was brought to the reader's attention that this model is still perfectible and the methods that were used to choose the hyperparameters could be improved by using a proper cross validation. We can also note that there is a quite important trade-off between the use of this model and a simple logistic regression in terms of computation time which needs to be considered depending on the accuracy required.

[5] 2.8 GHz Intel Core i7 with 16GB of RAM, no GPU support

REFERENCES

[1] Y. Kim, "Convolutional neural networks for sentence classification," https://arxiv.org/pdf/1408.5882v2.pdf, New York University, 2014.

[2] Y. Zhang and B. C. Wallace, "A sensitivity analysis of (and practitioners guide to) convolutional neural networks for sentence classification," https://arxiv.org/pdf/1510.03820v4.pdf, University of Texas at Austin, 2016.

[3] S. Lai, "Recurrent convolutional neural networks for text classification," http://www.aaai.org/ocs/index.php/AAAI/AAAI15/paper/download/9745/9552, National Laboratory of Pattern Recognition (NLPR), 2015.