Language Models for Twitter Sentiment Analysis
NLP1 - Project 4

Davide Belli 11887532

Gabriele Cesa 11887524

Linda Petrini 11858931

Abstract

In this report we describe our approach on Twitter Sentiment Analysis. Precisely, we focused on the first subtask of the 4th task of the SemEval-2017 competition, namely the sentiment classification of tweets. In our work we tuned pre-trained word embeddings to include sentiment polarity information through distant supervised learning and then we employed those embeddings in several neural models, trained with the human labeled dataset provided by the competition. In our work we also experimented with Recurrent Additive Network (RAN), a new model recently proposed in NLP, and compared its performances with many other standard models.

.

1 Introduction

Twitter Sentiment Analysis is a relatively new area of Natural Language Processing that deals with the task of classifying the sentiment polarity of messages written by Twitter users. Twitter is a very popular social network and may represent a small window on our society: it counts more than 300 millions active users every month and 500 millions tweets are posted every day[1]. This is a valuable source of information which can give an insight in people's beliefs and ideas. As a result, in the last years this task gained more and more importance, with applications in companies interested in public opinion about their products, in the monitoring of popular events and even in helping to forecast market movements.

Indeed, *SemEval*[2], an ongoing series of evaluations of computational semantic analysis systems, recently included sentiment analysis tasks among the set of problems they propose every years. Accordingly, in this work, we tried to tackle the first subtask proposed in the 4th task of *SemEval-2017*[3]: identify the overall sentiment of tweets within 3 classes (positive, neutral and negative).

Nowadays, state-of-the-art approaches to this problem make extensive use of deep learning techniques, in particular CNNs and LSTMs. Indeed, the winners of this year's edition (Cliche, 2017) did the same and we followed a similar strategy. These architectures are particularly appropriate for Natural Language Processing, since both are able to find some kind of short-distance relation between words. More in detail, CNNs process a sentence in one go but the convolution layers act in windows of fixed size, capturing in fact small contexts and, so, the relations between close words. On the other side, recurrent networks process sentences one word at the time, maintaining some memory of the previous words which enables them to exploit a longer context. Moreover, a variant of LSTM called Recurrent Additive Network (RAN) (Zettlemoyer, 2017), has been successfully proposed this year, especially applied for neural language models. In our work, we also tried to employ this new model, tailoring it to our task.

The code for the project can be found at[4] and at [5].

2 Problem

In order to evaluate our models we followed the SemEval competition instructions. For Task 4, the competition requires the contestants to fulfill five different subtasks, starting from a "simple" sen-

[1] https://www.omnicoreagency.com/twitter-statistics/

[2] http://alt.qcri.org/semeval2017/

[3] http://alt.qcri.org/semeval2017/task4/

[4] Sentiment Analysis: https://github.com/davide-belli/twitter-sentiment-analysis

[5] Distant Training: https://github.com/LindaPetrini/CNN_RAN/tree/davide

timent classification with three classes (positive, neutral, negative) and growing in complexity with more classes, topic-related classification and sentiment distribution modeling. We chose to focus on the first subtask, which is described in detail in (Nakov, 2017).

The organizers provide an human-annotated dataset, already divided in training, validation and test set. The classes distribution can be seen in Table 1. It is worth noticing the high class distribution imbalance, with almost four times more positive than negative tweets in the training set. This will be handled during training with apposite weights in the loss function.

The proposed evaluation measure is the *average recall*, or AvgRec. It is defined as the recall averaged between the classes *positive* (P), *neutral* (U) and *negative* (N) and it is computed as follows:

Table 1: SemEval Task 4 Dataset

	Negative	Neutral	Positive
Training Set	863	2043	3094
Validation Set	391	765	844
Test Set	3232	10344	7061

$$AvgRec = \frac{1}{3}(R_P + R_N + R_U). \quad (1)$$

R_i is the recall for class i calculated as:

$$R_i = \frac{M_{ii}}{\sum_j M_{ij}} \quad (2)$$

where M is the confusion matrix, i.e. $m_{i,j}$ is the number of datapoints of class i predicted to be in class j.

One of the main problems in this task is that the dataset is highly unbalanced, leading the network to be more sensible to one class. Standard accuracy relates closely to this issue. Conversely, the proposed evaluation measure takes this information into account, providing a reliable metric for model comparison.

3 Approach

Following the approach used by the first ranked team in last year's competition (Cliche, 2017), we decided to apply some of the most common deep learning techniques: Recurrent Neural Networks and Convolutional Neural Networks.

In order to process the tweets, it is necessary to encode and feed them to the neural networks. A very popular means of doing so is through embeddings, as firstly proposed by (Jauvin, 2003). Many datasets of pre-trained embeddings are available on-line, for example Google's *word2vec* and Stanford's *GloVe*. Starting with pre-trained embeddings generally increases performances for language modeling tasks, nonetheless the type of information usually encoded in the embeddings relates mainly the context of the words, than to the sentiment. For example, words as *good* and *bad* are generally close in the embeddings space since they appear in similarly constructed sentences, but obviously their sentimental meaning is completely different.

For this reason, as first proposed in (Qin, 2014), we exploited a larger but noisy dataset to fine-tune the embeddings to be more sentiment-related (*distant learning*), which will be useful later in the competition task.

In this section we are first going to present the models we employed and then illustrate in detail the training procedure we used.

3.1 Models Architecture

3.1.1 LSTM, RNN_TANH, RNN_RELU and GRU

We began the implementation of our models by working with the PyTorch tutorial Word Language Model [6]. This project already implemented basic neural networks in order to train embeddings for next-word predictions. We dug and experimented with their code, modifying it in order to accomplish the classification task of sentiment analysis.

All the models we implemented take as inputs batches of tweets, where each word is represented by its index in the dictionary. The first layer in the network is the Embedding layer, which looks up in a dictionary the embedding tensors for each word. Afterwards, embeddings are inputted in the hidden layer(s). Depending on the model chosen at initialization time, the hidden layers (we chose to use two hidden layers) may be RNN, LSTM or GRU layers. Furthermore, a Dropout layer is used to prevent co-adaptation of features after the encoding in embeddings and on the output of the hidden layer (Hinton et al., 2012). After the dropout, the ouput, sized *batch_size x tweet_len x hidden_units* is then passed through two Linear layers to shape it in matrices of second dimension 30 and then 3,

[6] https://github.com/pytorch/examples/tree/master/word_language_model

corresponding to the sentiment classes. Finally, a Softmax layer is applied to the outputs to turn them into meaningful probabilities.

3.1.2 LSTM Bidirectional

Inspired by the work of (Cliche, 2017), we decided to implement a bidirectional version of LSTM. The idea behind this network model is to also consider the words in a tweet in inverse direction, from the last word to the first one. In order to implement this version of LSTM, we copied and modified the initial part of our network model. The input of the LSTM consists of both the normal and the reversed tweet. Each of them is inputted in a separate series of hidden layers after being encoded into embeddings. Afterwards, the Tensors containing the outputs for the reversed sentence are reversed once again to arrange them in the correct initial order of words. Hence, the hidden layers of the two LSTMs are concatenated (word-wise) in one unique hidden state which is then connected through two simple fully connected layers (with 30 hidden neurons) to the three class output layer, which is finally normalized via the usual Softmax activation.

3.1.3 LSTM with Reverse hidden initialization

While we were implementing the bidirectional LSTM, we had an alternative idea about how to use the information provided by reading the tweet in reversed order when making predictions. Thus, we decided to include another version of LSTM in which the reversed tweet is used to initialize the hidden layer of the network. After the first pass is made, the output is discarded but the hidden layer status is used to initialize the layers in the next pass, where the tweet in the original order is actually used to make the final prediction.

3.1.4 RAN and RAN Bidirectional

Recurrent Additive Networks are a fairly new version of LSTM networks, proposed in (Zettlemoyer, 2017). The authors of the paper argue that it is possible to obtain results comparable with LSTM networks keeping the same gated structure, but applying non-linear activation functions only to the gates and not to the inputs. This modification leads to a simpler and faster-to-train model, since the output is now a weighted linear combination of the input values. Let $x = (x_1, \dots x_n)$ be the input, $h = (h_1, \dots h_m)$ be the output and $c = (y_1, \dots c_p)$ the hidden state. Then the model can be described by the following equations:

$$i_t = \sigma(W_{ic}\, c_{t-1} + W_{ix}\, x_t + b_i) \tag{3}$$

$$f_t = \sigma(W_{fc}\, c_{t-1} + W_{fx}\, x_t + b_f) \tag{4}$$

$$c_t = i_t \circ x_t + f_t \circ c_{t-1} \tag{5}$$

$$h_t = \sigma(c_t) \tag{6}$$

where i and f stand respectively for the input and forget gate. Our model was implemented starting from a PyTorch implementation of the paper's experiments. The original code can be found at this address[7].

The bidirectional RAN is obtained following the same approach used for bidirectional LSTM.

3.1.5 CNN

The Convolutional Neural Network we choose to employ is inspired by the one in (Cliche, 2017) too. It has the following structure: the first layer in the network is an Embedding layer, which has the same duty as in the LSTM model. Then, we consider 3 different kinds of kernels (varying in their size). A convolution is applied in this manner: for every different kernel size m the input goes through a 2D convolution with C channels and kernel of size $m \times embeddings\_size$. Afterwards, a non-linear function is applied (a *ReLu*) and finally a MaxPool layer which preserve only the maximum value from each of the channels. The resulting vectors (which are $3 \cdot C$) are concatenated, fed to a Dropout layer (with probability 0.5) and passed to a fully connected layer with the 3 outputs. The last step is then a Softmax activation. Due to the huge size of the dataset and the lack of processing power, during the embeddings distant training we used a smaller model than the ones used in (Cliche, 2017), with kernels of size 2, 3 and 4 and $C = 10$. Conversely, when we trained the final model on the not noisy dataset we could afford a larger network. Therefore, we used the CNN in that paper with kernels of size 5, 6 and 7 and $C = 200$ (the one the authors reported as performing better alone). Furthermore, in this last model there is an additional 30 neurons hidden layer between the concatenated outputs of the convolutions and the output layer (the dropout is used both before and after this hidden layer).

[7] https://github.com/bheinzerling/ran

3.2 Training Procedure

3.2.1 Parsing and Preprocessing

We applied the same procedure for preprocessing to both datasets. In contrast to usual text datasets (e.g. the Penn Treebank), tweets can be challenging for text analysis since the 140 characters limit forces the users to get creative with abbreviations, which makes building a language model more complex. To solve this issue, we decided to make the following substitutions:

- Tweets with special characters (non-ASCII) are removed
- Multiple spaces and tabs between words are replaced by single spaces
- Possessive and abbreviated negations are split (ex: *Julia's* becomes *Julia s* and *don't* becomes *don t*)
- Urls are substituted with the tag <URL>
- Emoticons are substituted in the following way:
  - ":) :] :D ..." are replaced with the tag <smile>
  - ":(:[:'(..." are replaced with the tag <sad>
  - ":O :/ :P ..." are replaced with the tag <funny>
- Hash marks and at signs are removed from hashtags and user mentions.
- Numbers are replaced with the tag <num>
- Symbols in $\backslash - \_ " \# @ ( ), ! ? * ; : \ . \& \sim$ are removed

Moreover, all words were lowercased and a number of <PAD> tags were added at the end to have a fixed length input.

3.2.2 Unsupervised learning

Due to the shortage of computational capacity, instead of training our own word embeddings, we decided to download a database of pre-trained word embeddings[8], containing 3 million words. These word embeddings are obtained using the word2vec models, two-layer neural networks that, given an input word, try to predict the surrounding window of context words.

It is worth noticing that a huge number of words in our tweets dataset could not be found in the pre-trained embeddings dataset, due to the presence of abbreviations, tags and other Twitter-specific features.

3.2.3 Pre-training Embeddings (Distant Training)

Given the pre-trained embeddings, we employed a CNN to add the sentiment information. We downloaded a large noisy dataset[9], which in turn is just the merger of two other datasets. As stated in the webpage, this dataset is not recommended to be used for a final model; however, it is suitable for this phase. Moreover, tweets in this dataset belong only to the two class positive and negative but the same kind of dataset have been used in the literature for this purpose. Indeed, our goal here is to capture sentiment polarity of words and to distance opposite sentiment words in the embeddings, hence we do not need neutral tweets.

The whole dataset contains more than 1.5 millions of tweets. Nevertheless, we have been able to employ only 500.000 of them due to shortage of memory in the hardware we have. Therefore, the first 250.000 tweets of each class have been preserved and we have used them all for training (we think that validation and testing are less relevant in this phase since we do not need to build a good predicting model here).

We train for 10 epochs (plus one initial epoch in which embeddings are frozen), using batches of size 50 and optimizing with Adagrad (learning rate of 0.01). In order to preserve the word embeddings from moving too far from the original vector (thus loosing much context-information), we freeze them for the first epoch, optimizing only the network parameters, and we use a dropout of 0.5. In order to verify that the trained embeddings learned successfully to adapt to the sentiment classification task, we expect that the distances between words similar for context but different for sentimental meaning increase. For example, we considered the distance of opposite words such as *happy* - *sad*, and *love* - *hate* to verify the improvement of the embeddings expressiveness during the training. To measure the distance, we employ both Euclidean distance and cosine distance. See ta-

[8] https://code.google.com/archive/p/word2vec/

[9] http://thinknook.com/twitter-sentiment-analysis-training-corpus-dataset-20

Table 2: Distance between word embeddings before and after 10 Epochs.

	Euclidean Dist.		Cosine Dist.	
	Ep. 0	Ep. 10	Ep. 0	Ep. 10
happy-sad	41.13	76.07	0.7838	0.9065
love-hate	35.71	63.53	0.7408	0.8697

ble 2 to see how distances change after training 10 epochs.

3.2.4 Sentiment Prediction (Supervised Training)

To complete the main task of the project, namely the classification of a tweet among *positive*, *neutral*, and *negative* classes, we decided to implement and compare performances of different Neural Network architectures. We started by implementing established models such as GRU, RNN (with either *tanh* or *ReLu* activations), LSTM and CNN, switching then to the newer RAN and modified version of networks such as Bidirectional LSTM and RAN or LSTM with a reverse initialization of the hidden layer. In order to train our models, we tried using both randomly or zero initialized embeddings and then using word2vec embeddings from Google after the training for sentiment prediction. The dataset was partitioned in batches of variable size (we experimented with sizes from 5 to 200 tweets) and sequence length equal to the number of words in padded tweets. The batchification process allowed us to effectively run fast parallelized experiments on the GPU. To evaluate the correctness of prediction at each step in the training we used Weighted NLLLoss (Weighted Negative Log Likelihood, Eq. 7) plus L2Loss (averaged element-wise distance, Eq. 8).

$$\text{loss}(\mathbf{y}, \textit{class}) = -\text{weight}_{\textit{class}} \cdot \ln(\mathbf{y}_{\textit{class}}) \quad (7)$$

where $\mathbf{y}$ is the 3-dimensional vector containing the predicted probability of a tweet to belog to each of the classes and *class* is the real class of that tweet.

$$\text{loss}(\mathbf{w}) = \frac{1}{N} \sum_{i=1}^{N} |w_i|^2 \quad (8)$$

The regularization term was weighted by an hyper-parameter λ (see Sec.4.1 regarding hyper-parameter tuning). To tackle the problem of unbalanced training sets provided in the competition, we introduce in the NLLLoss a weight for each class proportional to the inverse of the frequency of that class in the training set. This technique assigns higher penalty to errors when predicting infrequent classes. To avoid overfitting, beyond using a validation set and a regularization term in the function loss, we added a dropout layer into the model. The dropout is not used during evaluation and testing phase.

Moreover, we tested two different ways to use back-propagation on the fitness function to train the network. Initially, we back-propagated through the network only when the last word of each tweet has been forwarded into the net. Later on, we tried to back-propagate after each word was inputted into the model. Our idea was that it could speed up the learning in the model even if it is not only evaluating the tweet as a whole, but only a subsentence of it. As a result, with this approach we try to make our models predict the sentiment of a tweet as soon as possible. This can be particularly effective when introducing bidirectional networks and LSTM initialized with reversed tweets, because when the model is predicting the sentiment of a word it has already an idea of what is in the rest of he tweet.

It is important to point out that the hidden layer in our networks is reinitialized after each batch, because tweet predictions are independent of each other.

The problem of the exploding gradient in the recurrent networks is solved by normalizing the gradient when computing the loss for every batch using a clipping value of 0.25. To avoid having our models training also on padding words, we specify the padding index in the dictionary when initializing the Embedding layer in the networks. This results in the embedding at that index being a constant tensor of zeros. Another issue to consider is higher relevance is given to the last data-points in the dataset. As a consequence, if one class appears more often than others in the last elements of the training set, the model may be biased to predict that class too frequently (especially with high learning rates). To avoid this, we shuffle the dataset and recompute the batches at every epoch.

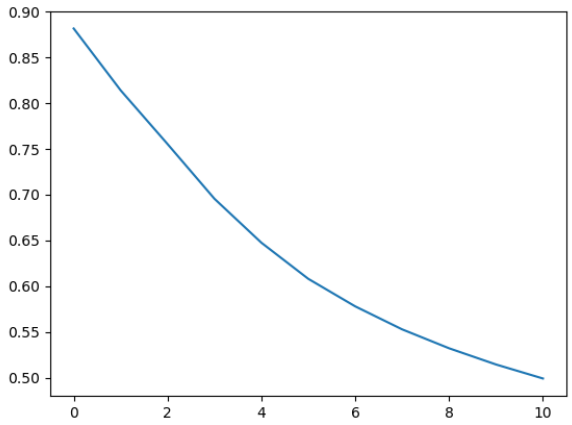
4 Experiments / Empirical evaluation

4.1 Tuning Hyper-parameters

The first step in executing the actual experiments involved finding the best configuration of parameters for the CNN to compute the pre-trained em-

beddings. Afterwards, we launched our model with the parameters described in Sec 3.2.3, effectively managing to improve the embedding performances, as represented by the decrease in the loss function plotted in Fig 1.

Figure 1: Evolution of NLLLoss while training Embeddings for 10 epochs.



Then, before running the final trainings to gather results from different models, we also executed many cross-validation experiment to find optimal parameters for our models. We tried different learning rates in a range from $1e-5$ scaling it times 2 up to 1 as largest value. Another important parameter to be tuned is the weight decay for the regularization term.

After trying different λ (weight decay) for regularization we decided to avoid including a regularization term in our loss function. Because this implies that the weight values might explode, we decided to employ early stopping to prevent overfitting. This means that we choose the best model on which to evaluate the test set by saving, among all epochs, the one which scores the best on the validation set. Moreover, a sufficient level of generalization is granted by the presence of dropout layers in our models. Other experiments were conduced to choose the best batch size, starting from smallest of dimension 5 to the largest, at 1000 tweets per batch. The tweet length is set to 40, which is the maximum length reached by tweets in our datasets. The same was applied to the number of units per hidden layer, while the size of embedding vectors were fixed to 300 in order to be compatible with our pre-trained embeddings which were initialized with word2vec embeddings. In Table 3 we present the parameters chosen to train and evaluate our models after this step of hyper-parameter tuning.

Table 3: Training parameters for the models, with brack-propagation after every word or at the end of the tweet.

	Back word	Back tweet
λ weight decay	0	0
learning rate	0.005	0.005
batch size	20	100
tweet length	40	40
epochs	30	300
embed. size	300	300
hidden layers	2	2
hidden units	200	200
grad. clipping	0.25	0.25
dropout	0.5	0.5

4.2 Experiment results

When an experiment is run, confusion matrices reporting classifications on training sets and validations sets are plotted in order to monitor convergence time and eventual overfitting. In the end, every model is tested in 4 different ways, combining the use or not of pre-trained embeddings and back-propagating after each word or only at the end of the tweet. The recall score for each model is reported in Table 4. For each column are highlighted the best and worst scoring models.

Table 4: Recall for Task4a of SemEval using different models

	Backprop word		Backprop tweet	
	no Emb	Emb	no Emb	Emb
GRU	0.4076	0.4417	0.4405	0.4670
RNN_TANH	0.3419	0.4537	0.3714	0.4158
RNN_RELU	0.4469	0.4383	0.3599	0.4522
LSTM	0.4217	0.4392	0.4312	0.4506
LSTM_BIDIR	0.4166	0.4828	0.4045	0.4588
LSTM_REV	0.4544	0.4753	0.4223	0.4482
RAN	0.4814	0.4883	0.4458	0.4698
RAN_BIDIR	0.4814	0.4855	0.4483	0.4452
CNN	—	—	0.3276	0.3317

In most cases we can see better performances when employing pre-trained embeddings. For LSTMs and RANs, it is also true that back-propagating after each word produces better results.

Overall, RAN models are able to obtain up to 5% better recall values when compared to other

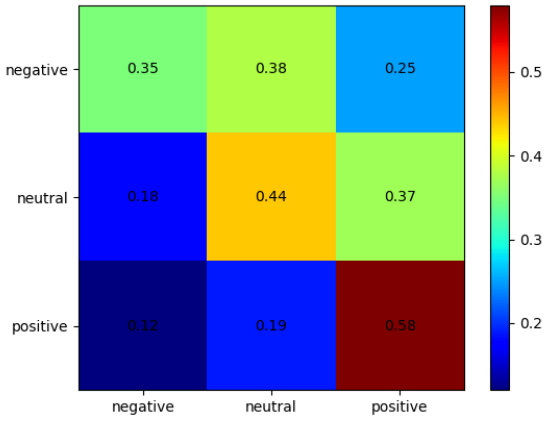
models. Employing RNNs for sentiment analysis, on the other way, seems to produce worse results. We think this might be explained by the difficulty to properly train simple RNNs and by their reluctance to learn long-term dependencies. In the last days before the deadline we also tried to test our CNN (modified for this purpose) on the SemEval dataset. Because its behaviour is different from previously considered recurrent network, it required different parameter tuning to work effectively but we didn't manage to find remarkable results. Anyway, we decided to include it in our report for completeness with respect to the project we are handing in.

To have a better visualization about how a model performs, we can consider the resulting *normalized* confusion matrix on the test set. For example, in Fig. 2 are plotted the performances of the LSTM with pre-trained embeddings while back-propagating at the end of the tweet. The labels on the rows describe the true sentiment, while the outputs of the model are represented in the columns. This confusion matrix is *normalized* with respect to the sum of the rows, namely the number of tweets belonging to each class (See Table 1 for values). As a result of this normalization, each cell contains the probability of our model to predict a certain sentiment given the true sentiment. The values on the diagonal cells are True positive, True neutral and True negative, and they represent the Recall for the respective classes. The average recall (Eq. 1), that is the average of the diagonal cells, expresses the performance of our model. The optimal confusion matrix, then, would be a diagonal matrix.

Comparing our results to the ones obtained by (Cliche, 2017), we notice substantial differences. The main reasons we use to explain this are that, for hardware limitation, we used a 200-times smaller dataset to pre-train embeddings, and that the authors added previous years' SemEval datasets to the training one for 2017. Moreover, they built 10 different CNNs and 10 different LSTMs and combined them in an *ensamble* to boost their performances.

Finally, we find it interesting to see how our model behave on tweets we create. We wrote a small script to test saved models on any tweet, reporting probabilities and predictions after every word is taken as input by the neural network. In Fig. 3 we can see predictions made with

Figure 2: Normalized confusion matrix of LSTM Model with pre-trained embeddings and backprop after tweet.

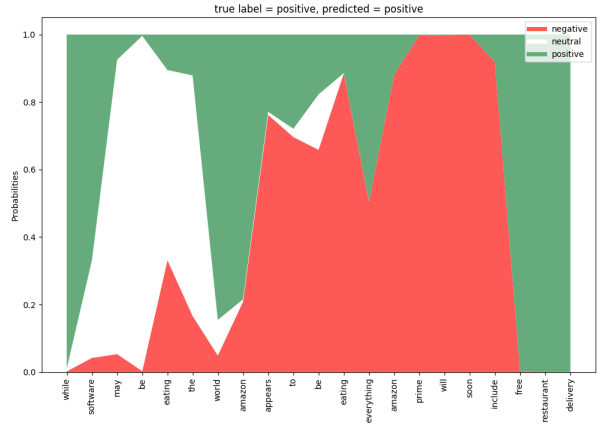


the RAN model with pre-trained embeddings and back-propagation after each word for the tweet:
"While software may be eating the world, amazon appears to be eating everything. Amazon prime will soon include free restaurant delivery!"
It is worth noticing the impact of the words: *amazon*, *prime* and *free*. The provided dataset contains a number of tweets related to the topic Amazon, hence the embeddings for the related words have quite a strong (apparently negative) influence. The word *free* on the other hand seems to have a rather positive influence, as one might expect. Similar behavior is confirmed when testing on many different sentences, also including emoticons. Adding padding tags in the end of a tweet, as we would expect, doesn't change predictions with respect to the one at the end of the sentence.

5 Discussion and Conclusions

In this report we presented and compared how different neural networks perform in the Twitter Sentiment Analysis task, in particular SemEval-2017 Task 4A. We noticed how it is not enough to only rely on a direct neural network approach in order to obtain good task score. Many improvements can be made. At first, we tried modifying normal NNs in order to amplify the information they are trained on, for example by considering the reversed tweets in LSTM_BIDIR, RAN_BIDIR and LSTM with reversed tweet for hidden layer initialization. Moreover, we noticed major boosts in our score by including preprocess-

Figure 3: Class probabilities, back-propagation after every word.



ing on the dataset in order to reduce noise (non-ASCII words, URLs, numbers, emoticons, symbols) and normalize words (capital letters, possessives, handles for references). This led to an increase of $3 \sim 5\%$ in the average Recall score. Finally, initializing the models with pre-trained embeddings, instead of random-initialized ones as default, can drastically change performances. Unluckily, this operation is computationally really heavy. Despite using a GPU[10] to improve computational time, the limited memory does not allow the use of datasets larger than 500 thousands tweets, and it takes many hours to train embeddings for few epochs.

One of most interesting insights we got from this project was the use of RAN for Sentiment Classification. Due to the fact that this model was introduced just a few months ago, the possible applications are still being explored and we could not find any paper employing RAN for this task. Indeed, our results show that this new model could outperform LSTMs in this particular task (though in (Zettlemoyer, 2017) the authors claim to get performances similar or slightly worse than LSTM).

For future work, it would be interesting to properly train embeddings with larger datasets on effective GPU servers, to see if this can properly improve the performances of our models. Furthermore, it would be worth trying to implement ensemble models of different neural networks, comparing and analyzing their scores with respect to individual models.

[10]GeForce GTX 960M, 4Gb memory

6 Team responsibilities

6.1 Davide Belli

- Adaptation of LSTM, GRU and RNN from word_language_models to sentiment analysis.
- Implementation of LSTM-BIDIR and LSTM-REV.
- Confusion matrix plotter
- Tweet preprocessing
- Tweet predictor

6.2 Gabriele Cesa

- Adaptation of CNN for sentiment classification
- Debugging and code review/refactoring
- Paper research

6.3 Linda Petrini

- Implementation of CNN for embeddings distant training
- Adaptation of RAN for sentiment analysis
- Implementation of RAN-BIDIR
- Paper research

References

Mathieu Cliche. 2017. Bb_twtr at semeval-2017 task 4: Twitter sentiment analysis with cnns and lstms. *CoRR* abs/1704.06125. http://arxiv.org/abs/1704.06125.

Geoffrey E. Hinton, Nitish Srivastava, Alex Krizhevsky, Ilya Sutskever, and Ruslan Salakhutdinov. 2012. Improving neural networks by preventing co-adaptation of feature detectors. *CoRR* abs/1207.0580. http://arxiv.org/abs/1207.0580.

Yoshua Bengio Rjean Ducharme Pascal Vincent Christian Jauvin. 2003. A neural probabilistic language model. *Journal of Machine Learning Research* .

Sara Rosenthal Noura Farra Preslav Nakov. 2017. Semeval-2017 task 4: Sentiment analysis in twitter .

Duyu Tang Furu Wei Nan Yang Ming Zhou Ting Liu Bing Qin. 2014. Learning sentiment-specific word embedding for twitter sentiment classification. *Proceedings of the 52nd Annual Meeting of the Association for Computational Linguistics,* .

Kenton Lee Omer Levy Luke Zettlemoyer. 2017. Recurrent additive networks. *CoRR* abs/1705.07393. http://arxiv.org/abs/1705.07393.