UNIVERSITY OF PISA

Sentiment Analysis on Tweets: Message Polarity Classification

Human Language Technologies project report

Amendola M. Cornacchia G. Salinas M.L.

Convolutional Neural Networks (CNNs) have recently achieved remarkably strong performance on sentence classification task [Attardi and Sartiano, 2016] [Kim, 2014] [Severyn and Moschitti, 2015] and extensive analysis have been done in order to distinguish between important and inconsequential design decisions [Zhang and Wallace, 2015]. Furthermore, in 2018, a new language representation model called BERT [Devlin et al., 2018] improved previous state-of-the-art scores in several tasks, including sentiment analysis. Thus we validate and evaluate different models, combining CNN and BERT methodologies, to tackle the SemEval-2017 Sub-Task 4A [Rosenthal et al., 2017]: message polarity classification; that is to decide, given a message, whether it expresses positive, negative or neutral sentiment. In the end our experiments confirm that contextual embeddings - and in general the BERT model - result in critical improvements and reach state-of-the-art performances.

Key words: Human Language Technologies, Sentiment Analysis, CNN, BERT

CONTENTS

1 INTRODUCTION

State-of-the-art approaches for message polarity classification involve Deep Learning architectures using Convolutional Neural Networks (CNNs). We decided to combine these approaches with a recently proposed language model BERT in order to tackle a Sentiment Classification task: message polarity classification [Rosenthal et al., 2017]. Our analysis can be divided in three sub-experiments:

- starting from a simple baseline model [Zhang and Wallace, 2015] with a one-layer CNN we cross-validate different hyperparameters configuration in order to choose the most performing one
- we adapted, namely *fine-tuned*, the **B**idirectional **E**ncoder **R**epresentations from **T**ransformers (BERT) on our message polarity classification task
- we also tried another BERT approach that consits in extracting features vectors from a pre-trained model and use them as a contextualized input representation

Then we select the most performing model among the one validated in each sub-experiment and compare these three models with each other. In the end the BERT model that uses fine-tuning is the one that achieved state-of-the-art performances.

1.1 BACKGROUND AND PRELIMINARIES

The first model we choose to implement is a Neural Network with Convolutional layers. Convolution is a key notion in image processing but recently has been successfully used to tackle Sentiment Classification tasks. To use convolution we transform each tweet in a *sentence matrix* $\mathbf{S}$, the rows of which are word vector representations of each token.

If the dimensionality of the word vectors is d and the length of a given sentence is s, then $\mathbf{S} \in \mathbb{R}^{s \times d}$. We perform convolution on $\mathbf{S}$ via linear filters $F \in \mathbb{R}^{h \times d}$. We will refer to h as *kernel size* from now on. The output sentence $\mathbf{o} \in \mathbb{R}^{s-h+1}$ of the convolution operator is obtained by repeatedly applying the filter on sub-matrices of $\mathbf{S}$:

$$\mathbf{o_i} = \mathbf{w} \cdot \mathbf{S}[i : i + h - 1] \tag{1.1}$$

where $\mathbf{w}$ is the weight matrix that represents the filter, $i = 1 \dots s-h+1$, and $\cdot$ is the dot product between the sub-matrix and the filter. If we want to add a bias b and an activation function f, a feature c_i is then obtained as:

$$c_i = f(\mathbf{o_i} + b) \tag{1.2}$$

Hence a convolution layer is able to capture patterns in sequences; our model will use the output of such layer to predict the general sentiment of the input tweet.

The last two sub-experiments are based on BERT. It is a new language representation model designed to pre-train deep bidirectional representations from unlabeled text by jointly conditioning on both left and right context in all layers. BERT use a "masked language model" (MLM) pre-training objective that randomly masks some of the tokens from the input, and the objective is to predict the original vocabulary id of the masked word based only on its context. During pre-training, the model is trained on unlabeled data over different pre-training tasks. Several pre-trained models have been released; for our analysis we only need to consider the uncased-english ones: $BERT_{BASE}$ (12 layers, 768 hidden, 12 heads, 110M parameters) and $BERT_{LARGE}$ (24 layers, 1024 hidden, 16 heads, 340M parameters). We explored two strategies for applying pre-trained language representations to downstream tasks: feature-based and fine-tuning. With feature-based strategy we can use the pre-trained BERT to create contextualized word embeddings. Then we feed these embeddings to our existing model. For fine-tuning, the BERT model is first initialized with the pre-trained parameters and all of the parameters are fine-tuned using labeled data from the downstream tasks.

2 DATASETS

For our experiments we used the official SemEval2017 corpora[1]: it includes train, dev and test set from 2013 up to 2017. We decided to train and validate our models on the union of train and dev sets while leaving out all test[2] sets for risk evaluation. In this way we were able to assess the risk of our final model over 8 different test sets, as depicted in table 2.1.

	Dataset Name	Negative	Neutral	Positive	Total
Train&Val	twitter-2013train-A	1458	4,586	3,640	9,684
	twitter-13dev	340	739	575	1,654
	twitter-15train	66	253	170	498
	twitter-16train	2,043	863	3,094	6,000
	twitter-16dev	391	765	843	1,999
	twitter-16devtest	325	681	994	2,000
		3,360	8,772	9,108	21,826 (without dup 21,240)
Test	twitter-2013test-A	559	1,513	1,457	3547
	sms-13test	394	1,208	492	2,094
	twitter-14test	202	669	982	1,853
	twitter-14sarcasm	40	13	33	86
	ljourn-14test	304	411	427	1,142
	twitter-15test	365	987	1,038	2,390
	twitter-16test	3,231	10,342	7,059	20,632
	twitter-17test	3,972	5,937	2375	12,284
		9,067	21,078	13,881	44,028

Table 2.1: Statistics about train, validation and test sets.

DATA PREPROCESSING

As first step we merged all the dataset involved in the train&validation step, removed all duplicates and cleaned each sentence using `tweet-preprocessor`[3]. Once the data was merged and cleaned we tokenize each sentence using a Keras Tokenizer, then we looked at the distribution of the length of tweets in our train data.

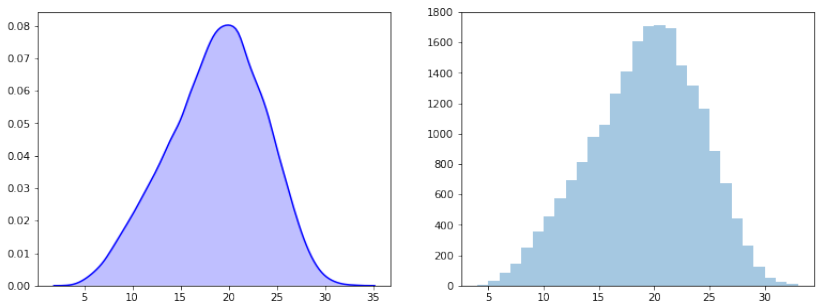

Figure 2.1: The distribution plot of the length of tweets in the train data and the relative histogram.

Figure 2.1 shows a mean length of 18.67 tokens per sentence, with values in [4,33] so we decided to pad each tweet to have a fixed-length of 40 tokens.

[1] http://alt.qcri.org/semeval2017/task4/index.php?id=download-the-full-training-data-for-semeval-2017-task-4

[2] http://alt.qcri.org/semeval2017/task4/data/uploads/semeval2017-task4-test.zip

[3] https://pypi.org/project/tweet-preprocessor/

3 MODELS

We tried CNN form [Zhang and Wallace, 2015] (CNN_{ZW}) and two different BERT approaches: fine-tuning and feature extraction from a pre-trained model. In this section we report more detailed information about the models and architectures used in our experiments.

3.1 CNN_{ZW}

The CNN classifier is based on a Deep Learning architecture, takes in input padded and tokenized sentences and consists of the following layers:

1. a lookup layer that maps every input token into a vector in the word embedding space
2. a convolutional layer with ReLu activation
3. a maxpooling layer
4. a dense layer with ReLu activation
5. a softmax layer

We used multiple kernel size and number of filters to learn complementary features from the same regions. A pooling function is thus applied to each feature map to induce a fixed-length vector. A common strategy is *1-max pooling* [Zhang and Wallace, 2015], which extracts a scalar from each feature map. The outputs generated from each filter map are then concatenated into a fixed-length, "top-level" feature vector, which is then fed through a softmax function to generate the final classification. At this softmax layer, one may apply dropout as a means of regularization [Hinton et al., 2012].

The code in figure 3.1 has been extracted from the module `run_cnn.py` in our github repo; `create_model(...)` is the function called to create the CNN and implements all our architectural decisions.

```
def create_model(kernel_size = (2, 3, 4), n_filter = (100, 100, 100),
      dropout = 0., activation = 'relu'):
  #Input layer
  tweet_input = Input(shape=(MAX_SEQUENCE_LENGTH,), dtype='int32')
  #Lookup layer
  tweet_encoder = Embedding(embedding_matrix.shape[0],
      embedding_matrix.shape[1], weights=[embedding_matrix],
      input_length=MAX_SEQUENCE_LENGTH,
      trainable=True)(tweet_input)

  conv_branches = []
  #Parallel Convolutional layers
  for k in range(len(kernel_size)):
    conv_branches.append(
        Conv1D(filters=n_filter[k],
          kernel_size=kernel_size[k],padding='valid',
          activation=activation,
          strides=1)(tweet_encoder))
    #1-max-pooling
    conv_branches[k] = GlobalMaxPooling1D()(conv_branches[k])
  #Concatenation of 1-max-pooling outputs
  merged = concatenate(conv_branches, axis=1)
  #Dense layer
  merged = Dense(256, activation='relu')(merged)
  #Optional Dropout
  merged = Dropout(dropout)(merged) if dropout > 0 else merged
  #Final softmax layer for prediction
  merged = Dense(3)(merged)
  output = Activation('softmax')(merged)

  model_ZW = Model(inputs=[tweet_input], outputs=[output])

  model_ZW.compile(loss='categorical_crossentropy',
           optimizer='adam')
  return model_ZW
```

Figure 3.1: CNN python code in `run_cnn.py`.

3.2 BERT

All the code that implements BERT models and approaches can be found on GitHub[4]. Next, in the document we will refer to BERT fine-tuning and feature extracion experiment respectivly as *BERT-ft* and *BERT-emb*.

FINE-TUNING

For fine-tuning experiments we decided to use only $BERT_{BASE}$ due to the lack of resources for the elaboration of $BERT_{LARGE}$. Both experiments have this parameter configuration:

Batch size	Sequence length	Dropout	Epochs
32	40	0.1	2

Table 3.1: Fine-tuning Model parameters

The first experiment consists on a single softmax layer on pre-trained BERT. For this purpose, we need a fixed dimensional representation of the segment. In fact, we pool the model by simply taking the hidden state corresponding to the first token as shown in Figure 3.2. The result has shape (batch_size, hidden_size). The code released by Google already contains the implementation of the model in the `run_classifier.py` module.

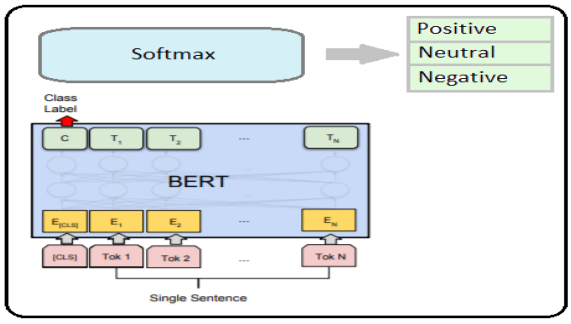

Figure 3.2: Fine-tuning model with softmax layer

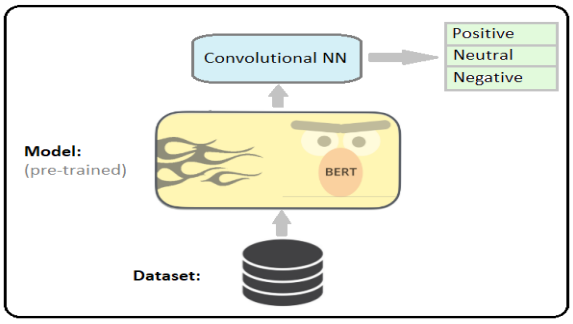

Figure 3.3: Fine-tuning model with CNN

In the second experiment we tried to put our CNN on top of BERT. Here we need a different representation: we want to use the token-level output. We don't pool the model, but we call the function

[4] Official BERT repository from Google Research `https://github.com/google-research/bert`

`get_sequence_output()` whose output has a shape `(batch_size, sequence_length, hidden_size)`. The CNN has the same parameters as the baseline in Table 4.2.

USING BERT TO EXTRACT FIXED FEATURE VECTORS

In certain cases, rather than fine-tuning the entire pre-trained model end-to-end, it can be beneficial to obtained pre-trained contextual embeddings [Devlin et al., 2018], which are fixed contextual representations of each input token generated from the hidden layers of the pre-trained model.

We extracted feature vectors for each token from the last hidden layer of both $BERT_{BASE}$ and $BERT_{LARGE}$ snapshots and used these contextualized embeddings in CNN_{ZW}. The `extract_features.py` implements the feature extraction from a pre-trained BERT checkpoint.

While there are major computational benefits to pre-compute an expensive representation of the training data once and then run many experiments with cheaper models on top of this representation, the output file produced by `extract_features.py` were very large, so in order to obtain "BERT-embedded" train data we had to split our dataset. Once all the transformed partitions were merged back the data looked different from the one we use for CNN_{ZW}: the input of the convolutional NN are tokenized sentences, that are vectors of integers, while in the BERT-emb approach the network takes in input directly a sentence matrix; thus to perform our experiments we simply removed the `tweet_encoder` layer from the CNN code in figure 3.1.

4 EXPERIMENTS

In this section we show our experiments with CNN_{ZW}, BERT-ft and BERT-emb. We took into account the suggestion from experiments carried out by [Zhang and Wallace, 2015] for choosing the parameters's search space of the CNN, while no grid search was performed in the experiments with BERT. We validate each model in section 4.1 and 4.2 with 3 repetition of a 5-fold cross validation and collect mean and standard deviation for three scores that, in order of importance, are: *f1-score*, *macro-averaged recall* and *accuracy*; these are the official scores used across the various reruns of the SemEval Subtask on sentiment classification [Rosenthal et al., 2017]. All the tables in this section are sorted on f1-score. The experiments were run on a linux server with an nVIDIA Tesla K40 accelerated GPU.

4.1 CNN_{ZW}

In order to give a reference for the CNN results, we consider as a baseline the architectural decision and hyperparameters of the CNN used by [Zhang and Wallace, 2015] for sentence classification, with small variations, described in table 4.1. while in table 4.2 we report the CNN_{ZW}'s validation scores.

At this point we now consider the effects of different hyperparameter settings, holding all the others constant. The validation scores for each configuration θ_i will be compared with the baseline in order to see if θ_i improves it.

EFFECT OF EMBEDDINGS

We first analyse the effect of different input representations. We used Glove [Pennington et al., 2014] and Google Word2Vec [5].

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

Hyperparameters	Value
Embedding	GloVE Twitter
Kernel size	(2, 3, 4)
Number of filters	(100, 100, 100)
Activaction function	ReLU
Dropout rate	0.0
Batch size	32
Pooling	1-max pooling
Epochs	2

Table 4.1: Baseline configuration. The difference from the CNN_{ZW} are the input representation and the kernel size.

	F1-score	MAVG recall	Accuracy
CNN_{ZW}	0.436 $\pm$0.034	0.567 $\pm$0.011	0.612 $\pm$0.008

Table 4.2: Baseline configuration scores.

	Word2Vec	GloVe	GloVe Twitter
Number of tokens	100B	6B - 42B - 840B	27B
Vector dimension	300	300 - 300 - 300	200
Corpus	Google News	Common Crawl	Tweets

Table 4.3: Embeddings details

At the beginning of the analysis we decided to change the baseline configuration of the CNN_{ZW} setting as the default input representation the GloVe Twitter embedding, because we thought that an embedding trained over a corpus of two billions of tweets would give us a better representation of a tweet, capturing in some way its lessical propriety. All the experiments were conduced using *non-static* word vectors. As we can see from Table 4.4, that summarizes the results of these different input representations, this assumption was wrong: in fact all the others embeddings overscore the baseline in terms of F1-score.

	F1-score	MAVG recall	Accuracy
Word2Vec	0.456 $\pm$0.032	0.573 $\pm$0.013	0.609 $\pm$0.006
GloVe.6B300	0.445 $\pm$0.049	0.570 $\pm$0.016	0.608 $\pm$0.007
GloVe.42B300	0.442 $\pm$0.038	0.564 $\pm$0.018	0.606 $\pm$0.010
GloVe.840B300	0.441 $\pm$0.038	0.567 $\pm$0.015	0.608 $\pm$0.008
GloVe Twitter (baseline)	0.436 $\pm$0.034	0.567 $\pm$0.011	0.612 $\pm$0.008

Table 4.4: Embeddings scores.

EFFECT OF KERNEL SIZE

Using the same *modus-operandi* as [Zhang and Wallace, 2015], to find the best kernel size for our data, we first analyze the effect of using only one region, keeping constant the number of filters as default (100); so we perform several CV for the kernel size $\in [1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,20]$.

From the result in table 4.5 one can see that using a kernel size of 13, 20 and 5 the CNN performs better w.r.t. the baseline, even if we use only one filter. At this point we consider the effect of combin-

Kernel size	F1-score	MAVG recall	Accuracy
13	0.450 ±0.027	0.571 ±0.014	0.605 ±0.004
20	0.446 ±0.033	0.569 ±0.013	0.603 ±0.007
5	0.445 ±0.034	0.567 ±0.012	0.606 ±0.008
12	0.441 ±0.032	0.565 ±0.015	0.602 ±0.010
11	0.440 ±0.034	0.568 ±0.014	0.606 ±0.007
2,3,4 (baseline)	0.436 ±0.034	0.567 ±0.011	0.612 ±0.008

Table 4.5: Single kernel size scores

ing different kernel sizes, keeping again the default number of filters. We decided to combine filters with kernel size near to the three optimal values found in the previous step; we obtain an improvement in terms of F1-score MAVG recall.

Kernel size	F1-score	MAVG recall	Accuracy
(6, 7)	0.460 ±0.018	0.573 ±0.006	0.605 ±0.004
(6, 6, 7, 8)	0.459 ±0.027	0.574 ±0.011	0.605 ±0.006
(6, 7, 8)	0.457 ±0.026	0.576 ±0.011	0.609 ±0.006
(5, 5)	0.456 ±0.025	0.575 ±0.007	0.611 ±0.005
(3, 4, 5, 6, 7, 8)	0.456 ±0.034	0.575 ±0.012	0.609 ±0.007
2,3,4 (baseline)	0.436 ±0.034	0.567 ±0.011	0.612 ±0.008

Table 4.6: Multiple kernel size scores

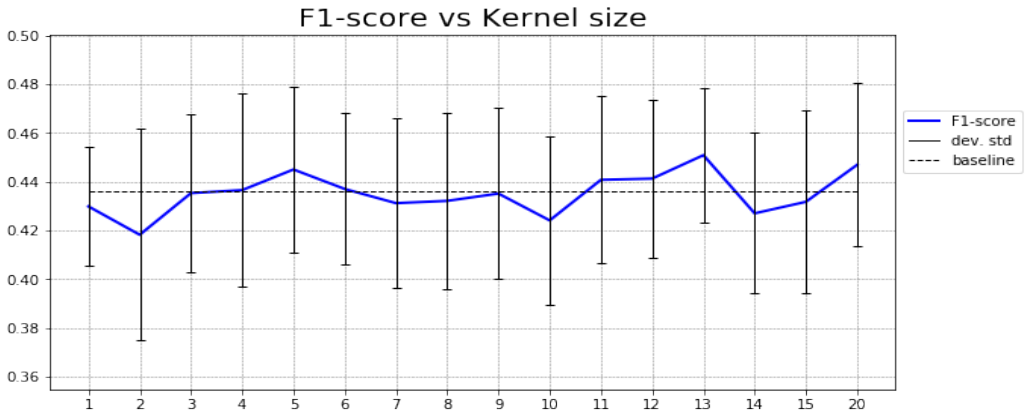

Figure 4.1: F1 vs Filter Size

Effect of number of filters

At this point we vary the number of filters for each kernel-size in the default configuration. We consider number of filters $\in [50, 90, 150, 200, 250, 300, 350, 400, 450, 500, 550, 600]$ and as we increase the complexity of the model we introduced a dropout term as mean of regularization.

The scores of top three models in Table 4.7 differ between each other by less than the 1% of their value, thus we can consider this three configuration as the best for the number of filters.

Number of filters and dropout	F1-score	MAVG recall	Accuracy
(200, 200, 200) dropout 0.3	0.456 ± 0.033	0.575 ± 0.010	0.609 ± 0.008
(150, 150, 150) dropout 0.0	0.455 ± 0.043	0.576 ± 0.017	0.608 ± 0.008
(400, 400, 400) dropout 0.0	0.454 ± 0.025	0.572 ± 0.009	0.606 ± 0.007
(250, 250, 250) dropout 0.3	0.450 ± 0.032	0.571 ± 0.013	0.606 ± 0.008
(200, 200, 200) dropout 0.5	0.450 ± 0.030	0.573 ± 0.011	0.612 ± 0.007
100, 100, 100 dropout 0.0 (baseline)	0.436 ± 0.034	0.567 ± 0.011	0.612 ± 0.008

Table 4.7: Number of filters scores

Effect of activation function

We consider all the activation functions of Keras, the one used in the default configuration is ReLU.

Kernel size	F1-score	MAVG recall	Accuracy
ELU	0.459 ± 0.030	0.573 ± 0.012	0.605 ± 0.006
TanH	0.446 ± 0.032	0.571 ± 0.010	0.611 ± 0.005
SoftSign	0.443 ± 0.030	0.570 ± 0.015	0.609 ± 0.004
SeLU	0.442 ± 0.039	0.566 ± 0.017	0.606 ± 0.007
Linear	0.436 ± 0.030	0.564 ± 0.012	0.607 ± 0.005
ReLu (baseline)	0.436 ± 0.034	0.567 ± 0.011	0.612 ± 0.008

Table 4.8: Activation function scores

As we can see from the table 4.8 the best activation function is ELU that overscores the second best (TanH) in term of F1-score of 0.013. One interesting result is that not applying any activation function (Linear) gives the same score of the baseline.

Effect of epochs

We analyze also the effect of the number of epochs during the training phase; we consider from 1 to 10 using also this time a dropout term.

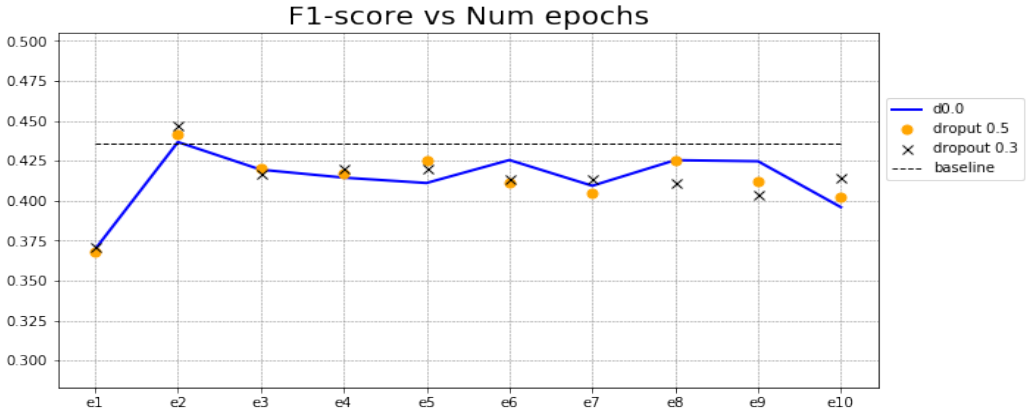

Figure 4.2: F1 vs Filter Size

The figure above shows that the optimal number of epochs in 2 (as the baseline) with a dropout value of 0.3. The table 4.9 shows that only with 2 epochs adding the dropout regularization increase the score of the model above the baseline.

Number of epochs	F1-score	MAVG reacall	Accuracy
2 and dropout 0.3	0.446 ±0.026	0.573 ±0.010	0.612 ±0.006
2 and dropout 0.5	0.441 ±0.038	0.572 ±0.013	0.614 ±0.005
6 and dropout 0.0	0.425 ±0.020	0.551 ±0.008	0.589 ±0.005
8 and dropout 0.0	0.425 ±0.030	0.551 ±0.010	0.589 ±0.008
8 and dropout 0.5	0.424 ±0.027	0.546 ±0.011	0.580 ±0.010
2 and dropout 0.0 (baseline)	0.436 ±0.034	0.567 ±0.011	0.612 ±0.008

Table 4.9: Number of epoch scores

OTHER HYPERPARAMETERS

We analyze also the effect of the batch size and the best scores were obtained by 16 and 32.

COMBINING THE HYPERPARAMETERS

In this second phase we try to combine the best hyperparameters found in the previous step in order to have some hints in the tuning of our CNN; performing also this time the same cross-validation. In the end the composition of the best hyperparameters didn't increase the scores, in fact, the best model is obtained in the first phase while we kept the default configuration changing only one hyperparameter. This can be explained by the fact that the default configuration was designed for Sentence Classification and thus it's tuned for a similar task. The best model we obtained, called ACS-CNN, was the one described in table 4.11.

Model	F1-score	MAVG recall	Accuracy
ACS-CNN	**0.456 ±0.025**	**0.575 ±0.007**	0.611 ±0.005
CNN_{ZW}	0.436 ±0.034	0.567 ±0.011	**0.612 ±0.008**

Table 4.10: ACS-CNN and CNN_{ZW} scores.

Hyperparameters	Value
Embedding	GloVE Twitter
Kernel size	(5, 5)
Number of filters	(100, 100)
Activaction function	ReLU
Dropout rate	0.0
Batch size	32
Pooling	1-max pooling
Epochs	2

Table 4.11: ACS-CNN configuration

4.2 BERT

This section describes our experiments with the BERT model. In the fine-tuned instance, we started from $BERT_{BASE}$ as pre-trained checkpoint and then we validate two different systems: one that has a softmax layer on top of the output of the last hidden layer for the `CLS` token, and the other using a CNN. Instead, in the feature extraction approach, we pooled out contextualized embeddings from

both $BERT_{BASE}$ and $BERT_{LARGE}$. Table 4.12 shows the validation results of BERT experiments compared with the baseline; top scores for each sub-experiment are in bold.

	F1-score	**MAVG recall**	**Accuracy**
$BERT_{FT-CNN}$	**0.691 ±0.024**	0.685 ±0.021	0.697 ±0.009
$BERT_{FT-Softmax}$	**0.691 ±0.023**	**0.687 ±0.019**	**0.698 ±0.008**
$BERT_{BASE}$	**0.556 ±0.044**	**0.641 ±0.023**	0.663 ±0.012
$BERT_{LARGE}$	0.550 ±0.074	0.638 ±0.030	**0.670 ±0.009**
CNN_{ZW}	0.436 ±0.034	0.567 ±0.011	0.612 ±0.008

Table 4.12: BERT experiments summary.

Both BERT sub-experiments outscore the baseline in every metric, with critical differences between the fine-tuned system and the one that uses static contextual embeddings. We can conclude that a model that uses BERT in one of its approaches is able to exibith strong generalization capabilities even with minimum tuning effort and thus can be expected to significantly improve both of an already existing model (feature extraction) or a simple one built from scratch to target a specific task(fine-tuning).

5 Results & Risk Evaluation

In this section we summarize the results of our experiments and evaluate the generalization capability of the best models found so far. In table 5.1 we compare the performances of the best scoring

	2013		**2014**			**2015**	**2016**	**2017**
	Tweet	**SMS**	**Tweet**	**Tweet Sarcasm**	**Live-Journal**	**Tweet**	**Tweet**	**Tweet**
ALL NEGATIVE	0.136	0.158	0.098	0.317	0.210	0.132	0.135	0.244
ALL NEUTRAL	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000
ALL POSITIVE	0.294	0.190	0.346	0.277	0.272	0.303	0.255	0.162
ACS CNN*	0.267	0.221	0.265	0.286	0.265	0.259	0.268	0.182
ACS BERT-EMB*	0.697	0.631	0.734	**0.533**	0.733	0.691	0.642	0.671
ACS BERT-FT*	**0.724**	**0.687**	**0.744**	0.479	**0.744**	**0.717**	**0.678**	**0.694**

Table 5.1: Scores on tests of the three best models

models in section 4, namely the one with *ACS* prefix in the table, with three baselines, on the test sets. As we expected BERT still outperforms CNN_{ZW}. Due to the validation results we decided to take only this model to the next phase of our analysis; that is comparing the generalization capacity of our best model (ACS BERT-FT) against the best scores of the previous years and two models from [Attardi and Sartiano, 2016]. From table 5.2 we can see how even a "simple" fine-tuned system with a softmax layer on top can achieve remarkable scores when compared to previous bests. As further risk assessment we collected and compared the performance of ACS BERT-FT for each class both on train and test.

In table 5.3 we present the breakdown of score among the three categories for the 2017 test data. The performance are pretty much the same across the different test sets: the model has stable performances across the different classes scores and validation data considered.

	2013		2014			2015	2016	2017
	Tweet	**SMS**	**Tweet**	**Tweet Sarcasm**	**Live-Journal**	**Tweet**	**Tweet**	**Tweet**
top scores 2013	0.690	0.685	-	-	-	-	-	-
top scores 2014	0.721	**0.703**	0.710	0.582	0.748	-	-	-
top scores 2015	**0.728**	0.685	**0.744**	**0.591**	**0.753**	0.648	-	-
top scores 2016	0.723	0.641	**0.744**	0.566	0.741	0.671	0.633	-
top scores 2017	-	-	-	-	-	-	-	0.685
UniPI	0.592	0.585	0.627	0.381	0.654	0.586	0.571	0.639
UniPI SWE*	0.642	0.606	0.684	0.481	0.668	0.635	0.592	0.652
ACS BERT-FT*	0.724_2	0.687_2	$\mathbf{0.744}_1$	0.479_{33}	0.744_4	$\mathbf{0.717}_1$	$\mathbf{0.678}_1$	$\mathbf{0.694}_1$

Table 5.2: Comparison between our unofficial submission, the one in [Attardi and Sartiano, 2016] and top scores from previous years.

	F1-score			MAVG recall		
	neg	**neutr**	**pos**	**neg**	**neutr**	**pos**
Train	0.772	0.765	0.843	0.808	0.743	0.853
Test	0.712	0.672	0.677	0.730	0.626	0.763

Table 5.3: ACS BERT-FT breakdown of score among the three categories for the 2017 test set

6 Conclusions

This report confirmed the effectivness of CNNs in sentiment analysis: with a minimal baseline model it is possible to achieve some results even on a such challenging task. We think that ,with ad additional effort in the tuning phase of the model, better results can be expected.

Furthermore, our experiments with the BERT language representation model highlighted its critical impact on our task performances: with a minimum coding effort it is possible both to fine-tune all the parameters of a "BERT-adapted-network" and extract the contextualized feature vectors of input tokens. BERT systems outperformed CNN_{ZW} on validation and test, and achieved an overall above average unofficial score over all the past runs of SemEval message polarity classification task.

All the developed code to implement the models in section 3 and run the experiments in section 4 can be found on GitHub[6].

[6] `https://github.com/hybrs/tweet-sentiment-analysis`

REFERENCES

[Attardi and Sartiano, 2016] Attardi, G. and Sartiano, D. (2016). Unipi at semeval-2016 task 4: Convolutional neural networks for sentiment classification. In *Proceedings of the 10th International Workshop on Semantic Evaluation, SemEval@NAACL-HLT 2016, San Diego, CA, USA, June 16-17, 2016*, pages 220–224.

[Devlin et al., 2018] Devlin, J., Chang, M.-W., Lee, K., and Toutanova, K. (2018). Bert: Pre-training of deep bidirectional transformers for language understanding. *arXiv preprint arXiv:1810.04805*.

[Hinton et al., 2012] Hinton, G. E., Srivastava, N., Krizhevsky, A., Sutskever, I., and Salakhutdinov, R. (2012). Improving neural networks by preventing co-adaptation of feature detectors. *CoRR*, abs/1207.0580.

[Kim, 2014] Kim, Y. (2014). Convolutional neural networks for sentence classification. *CoRR*, abs/1408.5882.

[Pennington et al., 2014] Pennington, J., Socher, R., and Manning, C. D. (2014). Glove: Global vectors for word representation. In *Empirical Methods in Natural Language Processing (EMNLP)*, pages 1532–1543.

[Rosenthal et al., 2017] Rosenthal, S., Farra, N., and Nakov, P. (2017). SemEval-2017 task 4: Sentiment analysis in Twitter. In *Proceedings of the 11th International Workshop on Semantic Evaluation*, SemEval '17, Vancouver, Canada. Association for Computational Linguistics.

[Severyn and Moschitti, 2015] Severyn, A. and Moschitti, A. (2015). UNITN: training deep convolutional neural network for twitter sentiment classification. In *Proceedings of the 9th International Workshop on Semantic Evaluation, SemEval@NAACL-HLT 2015, Denver, Colorado, USA, June 4-5, 2015*, pages 464–469.

[Zhang and Wallace, 2015] Zhang, Y. and Wallace, B. C. (2015). A sensitivity analysis of (and practitioners' guide to) convolutional neural networks for sentence classification. *CoRR*, abs/1510.03820.