



AI502 Deep Learning

Final Project - "Tweet sentiment extraction" Kaggle challenge

Cappa Victor
KAIST ID: 20206080
Academic Year 2019/20 Spring semester
KAIST School of computing
victor.cappa@kaist.ac.kr

Part 0 - "Tweet Sentiment Extraction" tasks and proposed solutions

The "Tweet Sentiment Extraction" dataset <https://www.kaggle.com/c/tweet-sentiment-extraction> consists in a collection of 27481 tweets, labeled with a sentiment (positive, negative or neutral) and characterized by a contiguous portion of the former tweet which is supposed to lead to the labeled sentiment. The goal of this Kaggle competition is to construct a model which can look at the labeled sentiment for a given tweet and figure out what word or phrase best supports it.

The evaluation metric for the challenge consists in the jaccard metric, which is provided in the challenge page, and which is supposed to measure the IoU (Intersection over Union) of the ground truth extracted tweet with respect to the one which is determined with our trained model.

Now we analyze a range of possible methods which could be adopted to solve the given task.

Proposed method 1 - RNN architecture

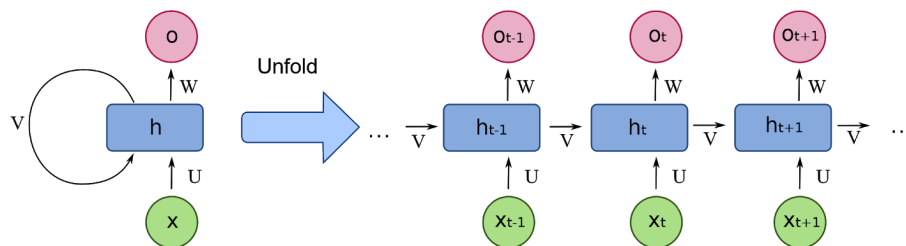


Figure1 -basic RNN picture

The first thing we have to consider is the sequential structure of our tweet data. We can exploit RNNs (Recurrent Neural Networks) to deal with the task at hand by feeding the network with the encoded tweets, and by stacking a FC layer on top of the hidden states of the RNN,. In this way we would be able to determine the starting and ending positions of selected text portion of each tweet. Different basic RNN modules can be used for this purpose, such as LSTM, GRU, stacked RNN structures or bidirectional RNNs. However, these models usually come without pre training, thus it could be necessary to have big datasets to train the models on, and moreover, classical RNN models don't have any sort of attention or self-attention included by default, thus they may be unsuitable to solve the task at hand.

Proposed method 2 - BERT based model (transformers)

A more powerful methodology consists in using transformers pretrained with BERT (Bidirectional Encoder Representations from Transformers). Transformer architectures already contain attention mechanisms such as self-attention, and multiple encoders inside of them, while being pre trained by default on massive amounts of data. Therefore, we exploit a very powerful deep learning technique called transfer learning, while also benefiting the complexity and capabilities of the model. However, training these models can require a lot of time and computational effort while running our experiments. This is the approach that I adopt to solve this task.

Proposed method 3 - Frequency based approaches

Since now, we just analyzed deep learning related methodologies to solve the given task, however, other less powerful but more computationally efficient approaches can be adopted, where we simply determine the frequencies of words in the selected tweet texts for each distinct sentiment, and we exploit this knowledge to determine keywords and words intervals in the test tweets.

Part 1 - Description of the proposed method and adopted methodology

In my proposed solution I solve the “Tweet Sentiment Extraction” task I use Bidirectional Encoder Representations from Transformers, or BERT, which is a revolutionary self-supervised pre training technique first released in 2018 that learns to predict intentionally hidden (masked) sections of text. Crucially, the representations learned by BERT have been shown to generalize well to downstream tasks, with highly successful results shown in transfer learning, allowing us to train our model on smaller training dataset while exploiting the previously learnt knowledge.

In particular, we are interested in Robustly optimized BERT approach, or RoBERTa, which consists in a further improvement of BERT. The used model is the one in the huggingface python library, and I focus my experiments on base-roberta, with the default configuration consisting in a 12-layer architecture, with a hidden size dimension of 768, 12-heads, and a total of 125 M parameters. RoBERTa (according to the Huggingface library) is implemented on top of the BERT-base architecture.

A simple graphical architectural explanation of Roberta is shown below:

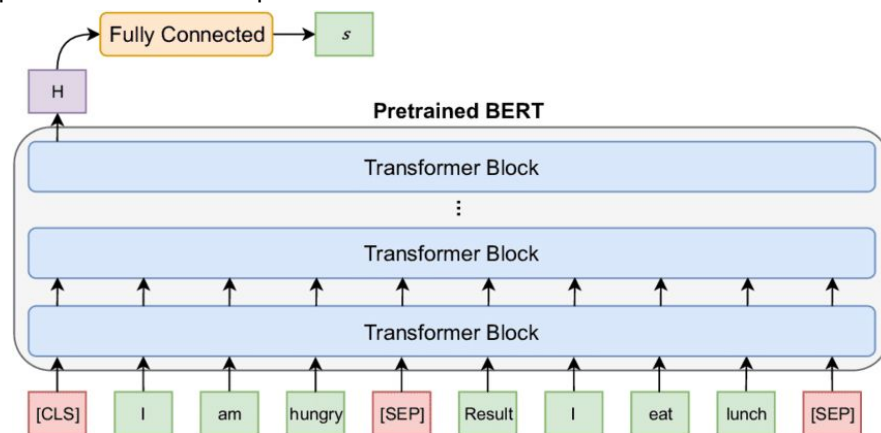


Figure2 - BERT architecture

Part 1.1 - Pytorch TweetModel

In my pytorch model, TweetModel, I have 192 output tensors for each transformer block layer, each one of those of dimension 768 (hidden-dimension size). One of the key methodologies I adopted consists in concatenating the last two hidden-dimensions for each output position in a single tensor. In this way I am able to achieve better performances. I also tried other configurations, such as summing all the hidden vectors for each output position, using only the last hidden-dimension, concatenating more than two hidden dimensions together, and summing the last few hidden-dimensions, but the chosen configuration outperforms all the previously described variants.

On top of these I stack a FC layer, after a Dropout layer with dropout of 0.05, allowing to obtain a two dimensional output tensor for each concatenated hidden dimension tensor, consisting in a score indicating the target_start score, and a target_end score, respectively indicating the start and end of the target string words. The input of the TweetModel model consists in a custom preprocessed and tokenized version of the tweet string (more information on this will be discussed in the pretraining and tweet encoding section). The output of the TweetModel model is two 192 dimensional tensors, respectively the start_logits and end_logits.

Part 1.2 - Loss function

The loss function should optimize over two different outcomes, computed on the start_logits and end_logits tensors, meaning the task at hand has a multi-task loss structure, and to solve it I simply sum the two losses and optimize over the composite loss. The cross-entropy loss is used to calculate the loss for each task.

In one of the experiments that I conducted I splitted the two tasks by training two distinct models, one to find the start_logits and another one to find the end_logits, such that the two tasks can be learnt

independently and they don't share common parameters. I concluded that splitting the two tasks doesn't improve the performances (nor training, nor final) by any means, but I get very similar results while training two different models, leading to higher computations for no benefits. Another approach I adopted consisted in weighting the two losses not equally, for example putting more emphasis on the first loss compared to the second one or vice versa, but there were no improvements of any kind in terms of jaccard score.

Part 1.3 - Training process discovery and Hyperparameter optimization

In this section I discuss hyper parameter optimization and the training procedure that I adopted while fitting the TweetModel to the task at hand. The key methodologies that I adopted are:

- learning rate decay schedule
- early stopping based on the evaluation set results
- weight decay on the model's trainable parameters
- dropout (see TweetModel model)
- Adam optimizer
- (attempt, but I didn't have time to complete it) ensemble learning

I run several experiments to better understand the training procedure and how the model behaves during training for different hyper parameter and architectural configurations.

I concluded that the model is very sensible with respect to the learning rate decay schedule, and in fact I tried both linearly decaying schedules with different configurations, sinusoidal decay schedules, no decay schedule and finally the decay schedule that I propose in my final version, consisting in decaying the learning rate by a factor of approximately 10 for each training epoch. The final learning rates that show the best training stability and final performances in term of loss decay over the epochs and validation accuracy is [5e-5, 1e-6, 5e-8, 5e-9, 5e-10, 1e-10, 3e-11, 5e-12] for a total of eight epochs.

Another interesting experiment I conducted has been using the RMSProp optimizer, but it didn't provide valuable improvements in the current configuration. I also wanted to try RAdam, but it looks like pytorch doesn't have this optimizer in its library yet.

Another important factor I noticed while training the TweetModel model is its eagerness to overfit the training dataset. This is a core issue, leading to poor overall performances, thus I adopted the early stopping technique in order to avoid this phenomenon and have a model that is able to maximize its generalization capabilities to the validation (and test) datasets. Other techniques which have been successfully adopted to avoid the overfitting problem in the proposed solution are dropout and weight decay.

However, overfitting remains a major core issue while training our model, letting me think that probably the dataset is too little to successfully train a high-performance TweetModel model, better able to solve the task at hand. Therefore, one of the best solutions includes increasing the number of training samples but this is not an easy, nor feasible solution, and/or introducing some data augmentation techniques, but it is not easy to achieve for this type of dataset with this type of task to solve. As a result, the best thing left to do is train our model with high quality data, meaning it is necessary to perform careful pre processing operations on the former training dataset.

Part 1.4 - Preprocessing

As I stated in the previous section, data preprocessing is very important and must be addressed carefully, especially with the given task. In the proposed solution, in the data preprocessing section, I first show some valuable dataset statistics and I determine that we have a total of 27481 observations with four dimensions (text_id, text, selected_text, sentiment). I encode the sentiment with the LabelEncoder class, and I drop all the records containing missing values. Then I generate a new dataset column for the preprocessed tweet text, which will consist in the training data for our model. This column contains the preprocessed version of our tweet text, where all the strings are lowercase and all the clusters of three or more contiguous equal letters are reduced to a single one. This choice

is motivated by the fact that there are a lot of tweets in the dataset containing words containing groups of three or more consecutive equal characters that should only have one, such as the words “sooooooooooooo bad!!!”, which should be “so bad!” for instance.


Finally, another important pre processing operation that I perform is lemmatization, using the WordNetLemmatizer class of the nltk package.

Part 1.5 - Tweet Encoding

Another core aspect of my implementation is the tweet encoding. This is another step that must be addressed carefully in order to successfully train the TweetModel. In my configuration I include in the encoded sequence a one-hot-encoding representation of the sentiment, consisting in three numbers where instead of having the 1 number I have 1313. In this way I am able to successfully encode the sentiment. In one of my experiments I tried to encode it as a single number in the input sequence but I get slightly lower performances compared to my final approach. I also tried to omit it, but the resulting performances are much lower. After that, I split the tweet in single words and I individually encode these by means of the tokenizer. Each tokenized word consists in a vector of numbers that is concatenated in the model encoding and separated by the tokenized space character. On top of the encoding vector I also generate a mask and type ids vector. In my final implementation I use the RobertaTokenizer tokenizer of the transformers library.

The reason why I split the tweet in different components (words) before tokenizing it is because I wanted a scalable solution to be able to generate also an offset vector, indicating, for each number in the encoding vector, the belonging of such number on each word of the tweet, so that I would be able to train the model with BCEWithLogitsLoss instead of the classical Cross entropy loss, but unfortunately I didn't have time to implement this feature, also because the training and evaluation code should be also modified accordingly to implement this new approach. I believe this new methodology can further improve overall performances and provide even more interesting results.

Part 3 - Experimental results and Kaggle submission score

1 submissions for Victor500		Sort by Most recent			
All	Successful	Selected			
Submission and Description		Status	Private Score	Public Score	Use for Final Score
kernel-500 (version 9/9) an hour ago by Victor500 From Kernel [kernel-500]		Succeeded 	0.65850	0.66103	<input type="checkbox"/>

With my current implementation I achieve a final public score of 0.66103. By analyzing the final output of my implementation I can see the resulting selected support phrases from the test tweets are reasonably good and in general valuable keywords are selected and useless ones are in general skipped. An example of this is for tweet number 4 “http://twitpic.com/4w75p - I like it!!” labeled as positive where “I like it!!” is selected, or for tweet number 6 “I THINK EVERYONE HATES ME ON HERE lol” labeled as negative where “HATES ME” is selected.

Part 4 - Conclusions

To conclude this report, I would like to say that working with BERT, and doing my experiments on a real world NLP task has been very interesting and I believe will be also valuable for my career. However, I would like to continue trying to improve my solution on this task, also after my final project delivery, because I still have other ideas about how to improve the model and consequently the final score that I didn't have time to implement and evaluate such as ensemble learning with cross validation, and training the model using a different encoding with an additional offsets vector and by using the BCEWithLogitsLoss instead of the cross entropy one.