490H Natural Language Processing Spring Term 2018

Coursework

Authors (CID): Kendeas Theofanous (01611093), Steven Battilana (01611771)

1 Introduction

Offensive language in social media exists in various forms, like images, audio and most importantly through text. The purpose of the coursework was to develop models to identify and categorise offensive written language found in tweets. In the following sections, we will outline some general remarks for the challenge and the three different tasks assigned. The code for the three tasks can be found on https://gitlab.doc.ic.ac.uk/kt3118/NLP_coursework.

2 General Remarks

The first thing we had to do to prepare for the tasks was to look at the *OffenseEval* raw data and understand its structure. The data set includes 13240 tweets, labelled for 3 different classification tasks. Therefore, for each task we had to collect a subset of the data with the relevant labels. After this, several steps had to be made as a general procedure before actually implementing deep learning architectures to solve problems.

2.1 Pre-processing

After getting the data into suitable data structures, we did some pre-processing in order to have a meaningful vocabulary before tokenization. Standard pre-processing methods involve removing any punctuation and stop words in the sentences and making everything lower case. We did some extra cleaning on the data, mainly to expand phrases that include apostrophes and cut down words, e.g "I'm" to "I am". Finally, we used a library to carry out stemming, i.e. reduced english words to their word stems, as we only want stem of words in the vocabulary used by our networks.

2.2 Tokenization and Word2idx

After pre-processing, we tokenized the tweets and then converted the vocabulary into indices. This means that after making every word in all tweets into separate tokens, a dictionary/vocabulary of unique words is created. Every token in the dictionary has a unique index purely on its first occurrence in the tweets. Then, all tweets are translated into a series of numbers/indexes representing the words that make the tweets up.

It's worth mentioning that after pre-processing, tokenization and word2idx, we obtain a matrix of tweets (list of indices of words) with dimensions 13240×82, where 82 is the longest tweet. The longest tweet before pre-processing had size 102, so we reduced the feature space by about 20%.

2.3 Embedding and *GloVe*

Throughout the tasks, we use word embedding. This technique is widely popular in Natural Language Processing tasks to map words to a vector representation. The aim of this representation is to quantify words in such a way that semantic similarities can be construed with other words in a large corpus without human categorization during a task.

We experimented with the popular GloVe word embedding to obtain vector representations for the words in our tweets. GloVe is a pre-trained vector representation which has multiple versions, depending on the desired length of representation. We chose to experiment with the GloVe 100 vector representation. We discuss our findings in the following sections.

2.4 Train-Validation-Test split

This standard procedure was used in all tasks. First, we split provided data into training and test, 90% and 10% respectively. Second, we split the training data again into a smaller training set and validation set, 90% and 10% respectively. The training and validation set were used during the training of the networks. A separate small test set was kept aside to test the performance of the networks on unseen data. Despite being a nice performance check, the number of training samples was greatly reduced, thus making the networks unable to learn as good as we would have expected.

3 Task A

In Task A, we were given a text which our system should categorise in offensive (label: OFF) and not offensive (label: NOT). Below we will describe how we developed our architecture for this task.

3.1 Architecture

For all models described below we used `Adam` with `learning_rate = 0.01` and `weight_decay = 1e-5` as our optimiser. Additionally, we moved the last activation function into the loss function using `nn.BCEWithLogitsLoss()` as it is numerically more stable as it uses the log-sum-exp trick for numerical stability. We use this function which first pushes the network output through the Sigmoid function and then computes the binary cross entropy between target and output.

Note, we pre-processed the given data as described above and created a batch generator for training. We ran our neural network for at most ten epochs in which the model gets trained on batches.

During training and testing we used the `nn.BCEWithLogitsLoss()`, F1 macro score, accuracy, precision and recall to monitor the progress.

Next, we described the models we used:

Long-short Term Memory Network: We started off by designing a naive architecture having only three layers: an embedding layer, a long-short term memory (LSTM) layer and a fully connected layer without activation function in the output layer. The LSTM output's dimension is being reduced by taking the mean over the dimension 1. We moved the last activation function into the loss function.

After running experiments it was clear that this model was underfitting and that we needed to adjust our network. Therefore, we add a convolutional layer usually used in imaging where it passes a filter over our sentences. As our research on the internet promises that our LSTM might be able to get some useful insight from the feature maps produced by the convolution. Next, we describe the aforementioned alteration in more details.

Convolutional Long-short Term Memory Network: This is an extension of the LSTM Network above. The first addition was a dropout layer with dropout probability of 20% after the embedding layer. Secondly, we added a Convolutional layer with a ReLU activation function. Lastly, we inserted a max-pooling layer just before the LSTM layer. The subsequent layers are the same as in the LSTM network.

The reasoning behind this architecture was inspired by the usage of Convolutional layers in imaging to try and get more feature maps that learn small patterns in images in separate feature maps. In our case the Convolutional layer would try to learn from the vector word representation some of the idiosyncrasies and underlying patterns that make up an offensive tweet.

The convolutional LSTM network performed a little bit better that the simple LSTM network but was still substantially lower than the provided baseline.

Long-short Term Memory Convolutional Network: Here we also start with the LSTM network as a base network and modified it. A Convolutional layer with a ReLU activation function and max-pooling layer were added between the LSTM and the output fully connected layer.

After trying to first have a Convolutional layer before the LSTM layer in an attempt to learn feature maps out of the tweets, we decided to try the opposite. We used the LSTM layer and its recurrent property to learn straight from the sequences of the cleaned tokenized tweets. After that, the Convolutional layer would collect the output of the LSTM layer and act as the main classifier.

This network performed best so far obtaining the following metrics:

	Loss	F1 (macro)	Accuracy	Precision	Recall
Training	0.187	0.7092	0.7915	0.8760	0.4326
Validation	0.392	0.4806	0.6138	0.3616	0.1557

Convolutional Long-short Term Memory *GloVe* Network: In this architecture we incorporated the *GloVe* embedding matrix in the `nn.nn.Embedding` layer.

Here, we hoped to get better results by using the *GloVe* embedded matrix, but it did not deliver the hoped for improvement in performance.

Convolutional Gated Recurrent Unit (GRU) Network: In this network we extended the LSTM convolutional network by replacing the LSTM by a GRU and adding a second fully connected layer right after the GRU.

Here, we tried to alter the network such that it should be able to fit a more complex model function by using the GRU and additional fully connected layer.

This performed even worse than the *GloVe* network.

After testing all five architecture, we decided to run the Long-short Term Memory Convolutional Network using the pre-processed official test dataset and to submit it. Sadly, our chosen network performed very poorly yielding $F1$ $(macro)$ $score = 0.4766284$.

A more detailed description of the models' layers with their weights can be found in the appendix.

4 Task B

In Task B we were given an offensive text and our system should categorise it into targeted at an individual (label: TIN) and untargeted (label: UNT). Below we will describe how we developed our architecture for this task.

4.1 Architecture

Following up from Task A, we started testing our most successful architecture, *Long-short Term Memory Convolutional Network*, on the training data with labels for Task B. After the same pre-processing, we removed all instances of the training data where the label was 'NULL', meaning that the tweets weren't offensive in the first place. This greatly reduced our training set, from 13240 instances to just 4400. The data set is highly unbalanced with a ratio of 9:1 for TIN:UNT. To tackle this imbalance, we used weights in the loss function in the same ratio. The same loss function and optimizer as Task A were used (`nn.BCEWithLogitsLoss()`, `Adam`). We also implemented a simple Early stopping technique to avoid over-fitting on the data. A check is done on the loss on the validation set, which if surpasses a certain tolerance of our choice, interrupts the training.

Convolutional Encoder and Long-Short Term Memory Network: In our attempt to improve the model from Task A and get better performance, we decided to build an Encoder followed by an LSTM layer and an output Fully Connected layer. The Encoder is made up of three Convolutional layers that increase the number of channels. Its purpose is to learn a latent representation of the data. After that, we would pass the data to the LSTM layer and do recurrent passes on the latent representation. The Fully Connected layer produced the output of the network.

We tested the various architectures explained in the previous task for this classification problem, but the one with the best perfomance was the Encoder with LSTM. The performance of this model is shown below:

	Loss	F1 (macro)	Accuracy	Precision	Recall
Training	0.158	0.4691	0.8840	0.8840	1.0
Validation	0.148	0.4734	0.8990	0.8990	1.0

5 Task C

In Task C task we were asked to design a system which identifies the offence target, that is categorise into individual (label: IND), group (label: GRP) and other (label: OTH) target. The data is even less for this task. After removing all "NULL" instances, only 3876 tweets remained. Below we will describe how we developed our architecture for this task.

5.1 Architecture

This is a multi-class classification problem, therefore we used a different loss function from the previous tasks. We used the Negative Log Likelihood loss function and our prediction for the samples would be the index of the maximum likelihood. This assumes that the output layer has three neurons. After the not very successful attempts from the previous tasks, we decided to choose a model that performed at least quite fast and above the provided baselines.

Long-Short Term Memory Convolutional Network: This task was solved in a similar manner as Task A. An LSTM layer followed by two Convolutional layers and an output Fully Connected layer were introduced. We also added a Dropout layer with a dropout probability of 50%.

The other networks explained in previous sections were tested, but the above network achieved a better performance:

	Loss	F1 (macro)	Accuracy	Precision	Recall
Training	0.256	0.2517	0.6070	0.3695	0.6070
Validation	0.387	0.2568	0.6264	0.3924	0.6264

The Early stopping technique implemented performs well here, where the validation loss increases above the tolerance and stops the training procedure after a few epochs.

6 Conclusion

After this initial, unfortunate, attempt to solve this coursework we would do the following as next step. Try to find a bigger data set to counterfeit the overfitting of the models. Furthermore, we would explore deeper architectures and different embeddings. Lastly, we would explore different ways of encoding the words during the pre-processing phase.

References

[1] Lecture slides

[2] Georgios K. Pitsilis, Heri Ramampiaro and Helge Langseth. *Detecting Offensive Language in Tweets Using Deep Learning*, Trondheim, Norway, 2018

[3] Viktor Golem and Mladen Karan and Jan Snajder, *Combining Shallow and Deep Learning for Aggressive Text Detection*, 2018

[4] Younghun Lee and Seunghyun Yoon and Kyomin Jung, *Comparative Studies of Detecting Abusive Language on Twitter*, 2018

7 Appendix

7.1 Task A Architecture Details

Long-short Term Memory Network:

```
Model_lstm(
      (embedding): Embedding(13828, 103)
      (lstm): LSTM(103, 20)
      (fully_connected): Linear(in_features=20, out_features=1, bias=True)
)
```

Convolutional Long-short Term Memory Network:

```
Model_conv_lstm(
      (embedding): Embedding(13828, 103)
      (dropout): Dropout2d(p=0.2)
      (convolution): Conv1d(82, 128, kernel_size=(103,), stride=(1,))
      (relu): ReLU()
      (max_pooling): MaxPool1d(kernel_size=3, stride=3, padding=0,
                          dilation=1, ceil_mode=False)
      (lstm): LSTM(1, 20)
      (fully_connected): Linear(in_features=20, out_features=1, bias=True)
)
```

Long-short Term Memory Convolutional Network:

```
Model_lstm_conv(
      (embedding): Embedding(13828, 103)
      (lstm): LSTM(103, 20, dropout=0.5)
      (convolution): Conv1d(82, 128, kernel_size=(3,), stride=(1,))
      (relu): ReLU()
      (max_pooling): MaxPool1d(kernel_size=3, stride=3, padding=0,
                          dilation=1, ceil_mode=False)
      (fully_connected): Linear(in_features=6, out_features=1, bias=True)
)
```

Convolutional Long-short Term Memory *GloVe* Network:

```
Model_conv_lstm_glove(
      (embedding): Embedding(13828, 103)
      (dropout): Dropout2d(p=0.2)
      (convolution): Conv1d(103, 5, kernel_size=(3,), stride=(1,))
      (relu): ReLU()
      (max_pooling): MaxPool1d(kernel_size=3, stride=3, padding=0,
                          dilation=1, ceil_mode=False)
      (lstm): LSTM(99, 20, num_layers=2)
      (fully_connected): Linear(in_features=20, out_features=1, bias=True)
)
```

Convolutional Gated Recurrent Unit (GRU) Network:

```
Model_conv_gru_2linear(
      (embedding): Embedding(13828, 103)
      (dropout): Dropout2d(p=0.2)
```

```
        (convolution): Conv1d(82, 128, kernel_size=(103,), stride=(1,))
        (max_pooling): MaxPool1d(kernel_size=3, stride=3, padding=0,
                                 dilation=1, ceil_mode=False)
        (gru): GRU(66, 20, dropout=0.5)
        (fully_connected0): Linear(in_features=20, out_features=10, bias=True)
        (relu): ReLU()
        (fully_connected1): Linear(in_features=10, out_features=1, bias=True)
    )
```

7.2 Task B

Convolutional Encoder and Long-Short Term Memory Network: `Encoder_lstm(`

```
        (embedding): Embedding(7630, 81)
        (dropout): Dropout2d(p=0.2)
        (conv1): Conv1d(81, 128, kernel_size=(3,), stride=(1,))
        (conv2): Conv1d(128, 256, kernel_size=(3,), stride=(1,))
        (conv3): Conv1d(256, 512, kernel_size=(3,), stride=(1,))
        (lstm): LSTM(75, 128, num_layers=20)
        (fc1): Linear(in_features=512, out_features=1, bias=True)
        (relu): ReLU()
    )
```

7.3 Task C

Long-Short Term Memory Convolutional Network: `Model_lstm_conv(`

```
        (embedding): Embedding(7164, 81)
        (dropout): Dropout2d(p=0.5)
        (lstm): LSTM(81, 46, num_layers=10)
        (conv1): Conv1d(81, 30, kernel_size=(3,), stride=(1,))
        (conv2): Conv1d(30, 5, kernel_size=(3,), stride=(1,))
        (fc1): Linear(in_features=5, out_features=3, bias=True)
        (relu): ReLU()
    )
```