Deep learning for NLP

Sentiment analysis on Twitter

Benoit Favre

22 Feb 2017

1 Introduction

In this tutorial, you will build a sentiment analysis system for Twitter. The task consists in predicting whether a tweet is positive, negative or neutral regarding its content.

Typically, sentiment polarity is conveyed by a combination of factors:

- An expression of subjectivity such as reference to first person pronouns or possessives
- The use of sentiment words such as "like", "best", "boring", "hated"...
- The modulation of these aspects by markers such as "hardly", "not"

Therefore, typical sentiment analysis systems rely on word lists for capturing subjectivity and sentiment bearing words, and word n-grams or parse trees for capturing modulation effects.

Since the quantity of data available for training is rather scarce, a number of approaches can be applied for making a system that generalizes well. For instance, words can be replaced by part-of-speech tags, or word classes. Sentiment-bearing word-lists can be extended through collocation statistics.

Tweets have a few specificities that make the sentiment analysis task more challenging. They contain very informal language with a lot of typos and colloquial language. Smileys can be an extra source of information but they are highly variable. Hashtags and user names are marked (along with other Twitter specific abbreviations). Word morphology can be modified to express sentiment (such as "looooooooool"). Therefore, formal linguistic analysis such as parsing and word lists are unlikely to be very helpful, but deep learning approaches can help dealing with that variability.

2 Reading the data

Here you will load a corpus of tweets annotated with four labels, `positive`, `negative`, `neutral` or `irrelevant`. The labels are related to a target (a

brand, etc). To begin with, we will ignore the target, and focus on the sentiment label and treat it as a 4-class classification problem. The corpus can be downloaded from `http://www.sananalytics.com/lab/twitter-sentiment/` but since they don't distribute the tweet text, we have put it for you in a file named `sanders-twitter-sentiment.csv`.

The file contains 5,513 annotated tweets in comma separated values format (csv). The columns are:

1. Example identifier in corpus
2. Twitter identifier
3. Time stamp
4. Text (that's the input for our classification task)
5. Sentiment label (that's what the system needs to learn)
6. Target

To load the corpus content, you can use the csv module from python:

```
import csv
with open('sanders-twitter-sentiment.csv', 'rb') as csvfile:
     reader = csv.reader(csvfile, delimiter=',', quotechar='"')
     for row in reader:
        print(row[3], row[4])
```

Next, tweets need to be preprocessed in order to normalize common phenomena such as hashtags, usernames, etc. For that part, we will use the `preprocess_twitter.py` module which is adapted from a script that can be found on the GloVe website[1]. It is a good idea to look at the normalization and generalization provided by this script, by trying it on various tweets. It contains a `preprocess(text)` function which returns normalized text.

So the first stage is to produce python list with all the preprocessed tweets (we can call it `texts`), and another with the labels (let's call it `labels`). We need to convert both to numpy arrays suitable for feeding them to a neural network.

For labels, it is rather straightforward. You can map the labels to integers, and then create a "one-hot" representation with the `to_categorical` function:

```
from keras.utils.np_utils import to_categorical

# assuming the labels list contains textual labels
label_mapping = {'positive': 0, 'negative': 1,
    'neutral': 2, 'irrelevant': 3}
labels = to_categorical(np.asarray([label_mapping[label]
    for label in labels]))
```

[1] `http://nlp.stanford.edu/projects/glove/preprocess-twitter.rb`

So the result should be a numpy matrix with one row per label encoded in one-hot representation.

For vectorizing tweet texts, first you can use the tokenizer provided in keras. It splits sentences into words and maps them to ids between 1 and the number of words in the lexicon. The tokenizer can be restricted to the most frequent words in the texts with the `nb_words` parameter.

Then, the next stage consists in padding the sequences with word 0 to a given maximum length (trimming them if they are longer than the provided `maxlen` parameter) with the `pad_sequences` function. It returns a matrix of shape (number of examples, max length) which contains integers (word ids). The word mapping can be found in `tokenizer.word_index`.

```
from keras.preprocessing.text import Tokenizer
from keras.preprocessing.sequence import pad_sequences

tokenizer = Tokenizer(nb_words=10000)
tokenizer.fit_on_texts(texts)
sequences = tokenizer.texts_to_sequences(texts)
data = pad_sequences(sequences, maxlen=32)
vocab = tokenizer.word_index
vocab['<eos>'] = 0 # add a word with id 0 for embeddings
```

If you followed the examples, the `data` and `labels` should be two numpy arrays of integers of shape (5513, 32) and (5513, 4). For monitoring performance in the experiments, we will use 4,000 examples for training and 1,513 for validation. Now is a good time to create arrays named `x_train`, `y_train`, `x_val` and `y_val`.

3 Word embeddings

Since the training dataset is small according to deep learning standards, it is unlikely that training embeddings on it will lead to good performance. Instead, you will use pretrained embeddings that can be downloaded from the web. The GloVe website[2] contains embeddings trained on 2 billion tweets with a vocabulary of 1.2 million word forms.

The good news is that the input text has been preprocessed with the same script we are using, so there is no mismatch between the preprocessing pipeline. The embeddings downloaded from the GloVe website are available in multiple dimensions, we'll arbitrarily select 100 for subsequent experiments. Since the embeddings are quite large, you can use the filtered down version from the file `glove.twitter.27B.100d.filtered.txt` which only contains the tokens seen in our data.

Off-the-shelf embeddings are generally distributed in a simple file format. On each line, the first token is the word and the next numbers represent the values of the word vector on each dimension.

[2] `http://nlp.stanford.edu/projects/glove/`

The `embedding.py` module helps loading an embedding file to a numpy matrix. It contains a `load` function which expects a dictionary associating words to ids, the dimension of the embedding and the name of the file containing the embeddings. It returns a numpy matrix with only the embeddings of the words from the vocabulary (as rows), and 0 for unknown words.

```
weights = embedding.load(vocab, 100,
    'glove.twitter.27B.100d.filtered.txt')
```

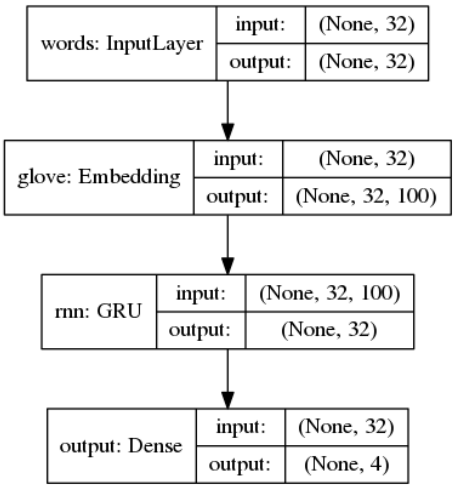
4 Recurrent neural networks

Our first model will be a recurrent neural network which reads a tweet word by word, mapping the word ids to their embedding representation, and then feeding the representations to a recurrent unit. The hidden state of the recurrent unit after reading all the words will be used as input of a dense layer with a softmax activation function to predict the probability of the four sentiment labels.

The embedding layer can be initialized with the embeddings we loaded from the web. We won't finetune them as we want to use them with words that are not in the training data. For recurrent unit, we will use Gated Recurrent Units (GRU) because they have less parameters and are faster to train than LSTM, but the later would also be a good choice. Their size is fixed to 64, but that's an hyperparameter that shall be explored.

```
from keras.layers import Embedding, Input, GRU, Dense
from keras.models import Model

# 100-dim embeddings initialized with GloVe,
# over sequences of size 32, and not fine tuneable
embedding_layer = Embedding(len(vocab), 100, weights=[weights],
    input_length=32, trainable=False)

sequence_input = Input(shape=(32,), dtype='int32')
embedded_sequences = embedding_layer(sequence_input)
x = GRU(64)(embedded_sequences)
preds = Dense(labels.shape[1], activation='softmax')(x)

model = Model(sequence_input, preds)
model.compile(loss='categorical_crossentropy',
    optimizer='Nadam', metrics=['acc'])

model.fit(x_train, y_train, validation_data=(x_val, y_val),
    nb_epoch=10, batch_size=64, shuffle=True)
```

Training in those conditions should get an accuracy around 80% on the validation set after a few epochs.

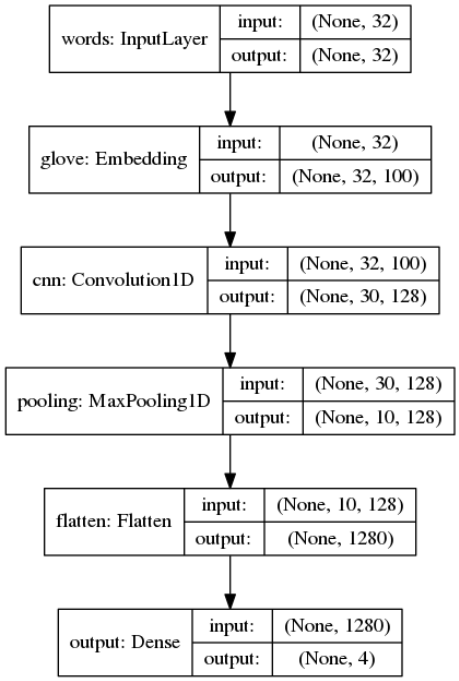


5 Convolutional neural networks

The recurrent model has the downside that its hidden state of fixed size might be suboptimal for building representations of the beginning of the input, and might be too order-dependent. An alternative to this model is convolutional neural networks. They consist of a convolution filter which is repeated over a window moving along the input. They act a bit like a bag of n-grams which can select relevant word n-grams. In the following example, we use 128 filters of size of 3 over the input, with global max pooling, and then flatten the results and pass it through a dense layer with a softmax activation to generate the probability distribution over labels.

```
from keras.layers import Conv1D, MaxPooling1D, Flatten

sequence_input = Input(shape=(32,), dtype='int32')
embedded_sequences = embedding_layer(sequence_input)
x = Conv1D(128, 3, activation='relu')(embedded_sequences)
x = MaxPooling1D(3)(x)  # global max pooling
x = Flatten()(x)
preds = Dense(labels.shape[1], activation='softmax')(x)

model = Model(sequence_input, preds)
```

The model is a bit faster to train than the recurrent one, but it leads to a lower accuracy. In particular, as accuracy on the training set raises to 99%, it reaches a high value on the validation set and then diminishes in later epochs. Clearly, the model is over fitting the training data, leading to poor generalization performance. It might be a good idea to extend that model with a regularization technique, such as for instance dropout.



6 Extensions

Getting the most out of the presented models requires some more work. Things to start with are:

- Make embeddings trainable
- Try different types and sizes of embeddings
- Change the size of the hidden units / convolution filters
- Add more layers, and even combine RNNs and CNNs. Note that increasing model capacity also requires more training data and might be affected by severe overfitting

Another extension to this work could be improve on the input representation instead of the model. For instance, one could use part-of-speech tagging to generalize words grammatical categories. The ARK part-of-speech tagger[3] would be a good starting point.

In a state-of-the-art model, one would want to build word embeddings that specifically represent sentiment polarity. As an example, you could reimplement/extend the system that finished 2nd at the SemEval competition in 2016. The code is available at `https://github.com/mrouvier/SemEval2016`.

[3] `https://github.com/brendano/ark-tweet-nlp`

A last commendable option is to read characters as input instead of words in hope of better modeling morphology.