CLS Pre-doc Summer School 2017
NLU with TensorFlow

Florian Schmidt, Yannic Kilcher
florian.schmidt@inf.ethz.ch, yannic.kilcher@inf.ethz.ch

July 3, 2017
ETH Zurich, Data Analytics Lab

In this exercise we will tackle *binary sentiment classification*, first using convolutional neural networks (CNNs), later using recurrent neural networks (RNNs) . Traditionally this is a supervised learning task. However, since hand-labeled data is a scarce resource, one often resorts to a weakly supervised setting [Kim14]. Our training data was generated by considering :-) and :-(smileys as a weak learning signal for sentiment. So strictly speaking, our task is to predict which of the two smileys has been removed.

Exercise 1 - CNN Sentiment Classification

Dataset We provide a large set of training tweets, one tweet per line. The dataset is available at

`https://polybox.ethz.ch/index.php/s/5hOF0DqGAmdUYSd`.

All tweets have been pre-processed so that all words (tokens) are lower-cased and separated by a single whitespace. Tweets in `*.pos.txt` files are positive tweets and those in `*.neg.txt` files are negative ones. We suggest you start testing your code with the smaller files and move to the `full` files once you are confident that your model is doing what it should.

Model Two dimensional filter operations have a long history in signal and image processing. Accordingly, CNNs have their origin in computer vision. Since the advent of neural word-embeddings [TRB10] [MSC+13] [PSM14] it has become common to treat words as d-dimensional vectors and sentences as concatenated vectors (e.g. as $d \times n$ matrices for a sentence of length n). Given this two-dimensional real-valued sentence representation, we can now apply the same convolutional techniques as in computer vision. Take a look at [Kim14] [SM15] [DLL+17] if you are interested in the model we implement here.

Our model is shown in Figure 1. The first layer embeds words into low-dimensional vectors. The next layer performs convolutions over the embedded word vectors using multiple filter sizes. For example, sliding over 3, 4 or 5 words at a time. Next, we max-pool the result of the convolutional layer into a long feature vector, add dropout regularization, and classify the result using a softmax layer. We will use only one channel for the convolution.

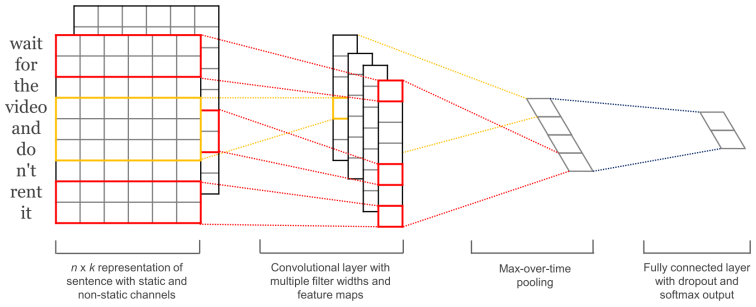

Figure 1: CNN model for sentiment classification (from [Kim14])

Getting started To get you up to speed quickly, we provide you with a code skeleton located in the `code` folder of the repository. This code loads the data, feeds it into a skeleton tensorflow model and performs a training loop. To be precise:

- The code loads the data, shuffles it and splits it into a training and a (1000 tweets) dev set.
- Then we pad the tweets with a <PAD> token to the length of the longest tweet. This allows us to use mini-batches since all the tweets will now be of the same size.
- We convert each word from every tweet to the corresponding index of the word in the vocabulary. Essentially, each tweet becomes a vector of integers.
- The first (embedding) layer of the CNN receives as input a batch of tweets in this format, and replaces each index with a word embedding for the particular word.
- These word embeddings can be randomly initialized, but typically for NLP tasks it is better to set their initial value to some pretrained word embeddings. For this you can either use the 300-dimensional word2vec embeddings or the 50-dimensional GloVe embeddings (depending on your RAM limitations). You can download the word embeddings you would like

to use from the Kaggle website and place them in the word-embeddings folder. In the sample scrip we have included a code that initializes the embedding matrix from the CNN. Make sure you have gensim installed[1].

- **Important**: In order to run the code, you need to provide the following program arguments

```
--training_file_pos /path/to../train_pos.txt
--training_file_neg /path/to../train_neg.txt
--embedding_file /path/to../glove_model.txt
```

Your task

1. Try to understand the code skeleton we provide, in particular take a look at `train_step` and `dev_step`. This is where data is fed into TF and results are retrieved. What is the difference between the two?
2. Complete the code for the CNN (in `text_cnn.py`). You find all necessary steps and some hints in the comments. If you get stuck, you can take a look at `text_cnn_solution.py`.
3. Start your program and start tensorboard[2]. If your code is correct, the accuracy on the dev set should go from 0.5 to something between 0.7 and 0.8 within about 2000 steps.
4. Perform the following experiments:
   (a) Investigate the impact of the pre-trained wordembeddings by commenting out the indicated line in `train_cnn.py`. Train with the same parameters and compare in tensorboard.
   (b) Change the parametrization of the convolution, e.g. the number of filters and their width. What gives you the best results? When is the model saturated?
   (c) Investigate the impact of the regularization by setting it to some very large and very small value. What do you observe?

Exercise 4 - RNN Sentiment Classification

As shown in the slides, RNNs can solve the same sentiment classification task by iteratively building up a summary of the sentence. With your CNN implementation at hand, it's actually only a few lines of codes to implement the RNN.

[1] `pip3 install --user gensim` should be sufficient. Also see `https://radimrehurek.com/gensim/install.htm`

[2] You need to start a separate python process e.g. using `python -m tensorflow.tensorboard --logdir /path/to/the/out/directory` in a terminal. Now you can access `http://localhost:6006` in your browser. If the page does not render properly, use Google's Chrome browser. See `https://www.tensorflow.org/get_started/summaries_and_tensorboard` for a detailed description of tensorboard.

The Model A recursive neural network, recursively computes a sequence of hidden states $h_1, h_2 \cdots \in \mathbb{R}^d$. Often this computation also depends on some input $x_1, x_2, \cdots \in \mathbb{R}^{d'}$ and we can write

$$f_\theta : \mathbb{R}^d \times \mathbb{R}^{d'} \to \mathbb{R}^d$$
$$h_t = f_\theta(h_{t-1}, x_t)$$

where θ is the parametrization of the network that we want to learn. In this case we can say that an RNN maps a sequence of inputs to a sequence of outputs while maintaining a state h_t. In our case the inputs will be words of a sentence and the state will be our sentence summary. At the end of the sentence, this summary should provide a fixed-size representation of the sentence that we can use to predict the sentiment. Just like the CNN max-pooling representation. Figure 2 shows the general RNN architecture.

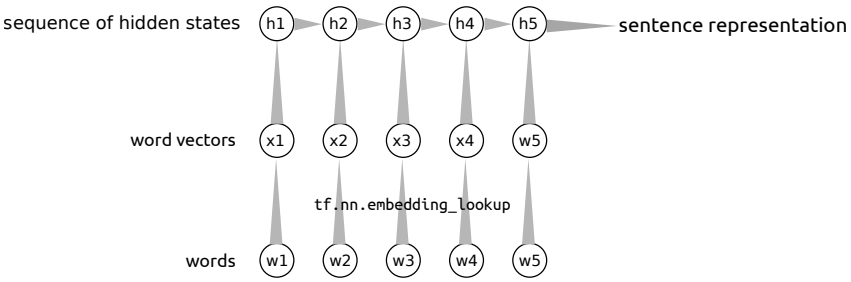

Figure 2: Abstract RNN architecture

Your task Create a new class `TextRNN` by copying the content of `TextCNN` from `text_cnn_solution.py`. Remove all the code between the `embedding` and the `dropout` scope. In between these two, we will define a new vector sentence-representation, now using an RNN instead of a CNN. Change the name of this representation from `self.h_pool_flat` to something more appropriate and adapt the size of the fully connected layer after the dropout accordingly. Also, you might want to change the constructor arguments of `TextRNN` to reflect the hyperparameters of the RNN.

1. Define the recurrent computation using the cell classes[3] of tensorflow. Use `tf.nn.rnn_cell.BasicRNNCell` for a start. The number of hidden units is a hyper-parameter. Try 64 as a start.

2. Use `tf.nn.static_rnn` to define your RNN given the cell from above. The returned final state is your new sentence representation. This function expects a *python list* of tensors as input, so you might want to take a look at `tf.unstack`.

[3] `https://www.tensorflow.org/api_docs/python/tf/nn/rnn_cell`

3. Run the model:

   (a) Open the graph in tensorboard (graph tab at the top). You should see a nested graphical representation of your network with boxes that can be expanded. Try to identify the RNN computation. Where is the chain of cells?

   (b) Use an LSTM cell[4] instead of the RNN cell. Compare the performance of the two by running with the same hyper-parameters.

   (c) Open the graph of the LSTM network in the tensorflow. Expand the box of one of the cells. Do you recognize the computation of the gates? Do you see the trick they use to compute all the gates at once?

   (d) Change the size of the hidden state size. How is performance affected?

References

[DLL+17] Jan Deriu, Aurélien Lucchi, Valeria De Luca, Aliaksei Severyn, Simon Müller, Mark Cieliebak, Thomas Hofmann, and Martin Jaggi. Leveraging large amounts of weakly supervised data for multi-language sentiment classification. *CoRR*, abs/1703.02504, 2017.

[HS97] Sepp Hochreiter and Jürgen Schmidhuber. Long short-term memory. *Neural Comput.*, 9(8):1735–1780, November 1997.

[Kim14] Yoon Kim. Convolutional neural networks for sentence classification. *arXiv preprint arXiv:1408.5882*, 2014.

[MSC+13] Tomas Mikolov, Ilya Sutskever, Kai Chen, Greg Corrado, and Jeffrey Dean. Distributed representations of words and phrases and their compositionality. *CoRR*, abs/1310.4546, 2013.

[PSM14] Jeffrey Pennington, Richard Socher, and Christopher D. Manning. Glove: Global vectors for word representation. In *Empirical Methods in Natural Language Processing (EMNLP)*, pages 1532–1543, 2014.

[SM15] Aliaksei Severyn and Alessandro Moschitti. Twitter sentiment analysis with deep convolutional neural networks. In *Proceedings of the 38th International ACM SIGIR Conference on Research and Development in Information Retrieval*, SIGIR '15, pages 959–962, New York, NY, USA, 2015. ACM.

[TRB10] Joseph Turian, Lev Ratinov, and Yoshua Bengio. Word representations: A simple and general method for semi-supervised learning. In

[4] See `http://colah.github.io/posts/2015-08-Understanding-LSTMs/` or the original paper[HS97] for an explanation of the recurrent computation.

Proceedings of the 48th Annual Meeting of the Association for Computational Linguistics, ACL '10, pages 384–394, Stroudsburg, PA, USA, 2010. Association for Computational Linguistics.