INF582
Lab session 5: Introduction to Deep Learning for NLP

Antoine Tixier, Konstantinos Skianis, Apostolos Papadopoulos, Michalis Vazirgiannis

March 15, 2017

1 Introduction

In this lab session dedicated to *Deep Learning for NLP*, we will get familiar with two of the most famous models from the state-of-the-art in Deep Learning: Convolutional Neural Networks (CNNs) and Long Short Term Memory networks (LSTMs), and see how we can apply those architectures to document classification. We will use Keras[1], a popular Python library for Deep Learning. Keras provides high-level building blocks for developing models, it does not handle itself low-level operations such as tensor products, convolutions and so on. Those are handled by its "backend" libraries[2], which currently include TensorFlow and Theano. Once you have installed Keras (through pip), you can select the backend of your choice by overwriting the `~/.keras/keras.json` file (of course the corresponding library will have to be installed on your machine as well).

2 IMDB Movie review dataset

In both sections of today's lab, we will be performing binary classification (positive/negative) on reviews from the Internet Movie Database (IMDB) dataset. This task is also known as *sentiment analysis*. The IMDB dataset contains 50,000 movie reviews, labeled by polarity (pos/neg). We will access the dataset through the `imdb.load_data()` method[3]. The data have already been partitioned into 50 % for training and 50% for testing. Furthermore, the reviews have been preprocessed and put in a format suitable to be passed to a neural network: each review is encoded as a list of word indexes (integers)[4]. Note that when you download the data for the first time, it might take up to a few minutes depending on your Internet connection, but it is just a one-time operation.

2.1 Binary classification objective function

The objective function that our models will learn to *minimize* is the *log loss*, also known as the *cross entropy*[5].

More precisely, in a binary classification setting with 2 classes (say 0 and 1) the log loss is defined as:

$$\text{logloss} = -\frac{1}{N}\sum_{i=1}^{N}\big(y_i \log p_i + \big(1-y_i\big)\log\big(1-p_i\big)\big) \tag{1}$$

[1] https://keras.io/
[2] https://keras.io/backend/#switching-from-one-backend-to-another
[3] https://keras.io/datasets/#imdb-movie-reviews-sentiment-classification
[4] Keras models are trained on Numpy arrays of input data and labels.
[5] https://keras.io/objectives/

Where N is the number of observations, p_i is the probability assigned to class 1, $\big(1-p_i\big)$ is the probability assigned to class 0, and y_i is the true label of the i^{th} observation (0 or 1). You can see that only the term associated with the true label of each observation contributes to the overall score. For a given observation, assuming that the true label is 1, and the probability assigned by the model to that class is 0.8 (quite good prediction), the log loss will be equal to $-\log(0.9) = 0.105$. If the prediction is slightly worse, but not completely off, say with $p_i = 0.6$, the log loss will be equal to 0.51, and for 0.1, the log loss is equal to 2.3. Thus, the further the model gets from the truth, the greater it gets penalized. Obviously, a perfect prediction (probability of 1 for the right class) gets a null score.

3 Paradigm switch

3.1 Feature embeddings

Compared to traditional Machine Learning models that consider core features and combinations of them as unique dimensions of the feature space, deep learning models often *embed* core features (and core features *only*) as vectors in a low-dimensional continuous space where dimensions represent shared latent concepts [4]. Furthermore, the coordinates of each core feature in that space (i.e., their embeddings) can be learned during training just like other model parameters. Initially, the feature embeddings have to be set to some values, which are typically random or obtained from pre-training[6].

3.2 Benefits of feature embeddings

The main advantage of mapping features to dense continuous vectors is the ability to capture similarity between features, and therefore to generalize. For instance, if the model has never seen the word "Obama" during training, but has encountered the word "president", by knowing that the two words are related, it will be able to transfer what it has learned for "president" to cases where "Obama" is involved, and behave accordingly. With traditional one-hot vectors, those two features would be considered orthogonal and predictive power would not be able to be shared between them[7]. Also, going from a huge sparse space to a dense and compact space reduces computational costs and the amount of data required to fit the model, since there are fewer parameters to learn.

3.3 Combining core features

Also, it is important to note that unlike what is done in traditional ML, combinations of core features are not encoded as new dimensions of the feature space, but as the *sum*, *average*, or *concatenation* of the vectors of the core features that are to be combined. Summing or averaging is an easy way to always get a fixed-size input vector regardless of the size of the training example (e.g., number of words in the sentence or document). However, both of these approaches completely ignore the ordering of the features. For instance, under this setting, and using unigrams as features, the two sentences "John is quicker than Mary" and "Mary is quicker than John" have the exact same representation. On the other hand, using concatenation allows to keep track of ordering, but *padding* and

[6] In NLP, pre-trained word vectors obtained with Word2vec or GloVe from very large corpora are often used.

[7] Note that one-hot vectors can be passed as input to neural networks. But then, the network implicitly learns feature embeddings in its first layer

truncation[8] need to be used so that the same number of vectors are concatenated for each training example. For instance, regardless of their size, every sentence can be transformed to have a length of n words: the longer sentences are truncated to their first (or last, middle...) n words, and the shorter sentences are padded with zero vectors for the missing words [14, 10].

4 Convolutional Neural Networks (CNNs)

4.1 Local invariance and compositionality

CNNs were developed in the field of Computer Vision to work on grids such as images. They are feedforward neural networks where each neuron in a layer receives input from a neighborhood of the neurons in the previous layer [11]. *Local receptive fields* allow CNNs to recognize more and more complex patterns in a hierarchical way, by combining lower-level, elementary features into higher-level features. For instance, edges can be inferred from raw pixels, edges can in turn be used to detect simple shapes, and finally shapes can be used to recognize objects. Furthermore, the model should be able to detect an object regardless of its position in the image. This property of the model is called *local invariance*. The ability to learn more and more complex features starting from small regions is called *compositionality*. Those two concepts are fundamental in understanding CNNs.

With that in mind, one can understand that CNNs can do well at NLP tasks such as sentence or document classification, for which higher-order features (n-grams) can be constructed from basic unigrams, and ordering is crucial locally ("not bad, quite good", "not good, quite bad", "do not recommend"), but for which the global location of the features within the sentence or the document does not really matter. Indeed, in trying to determine the polarity of a movie review, we don't really care whether "not bad, quite good" is found at the start or at the end of the paragraph. We just need to capture the fact that "not" precedes "bad", and so forth. Note that CNNs are not able to encode long-range dependencies, and therefore, for some tasks like language modeling, where long-distance dependence matters, recurrent architectures such as LSTMs are used instead (as we will see in the next section).

4.2 Convolution and pooling

The elementary construct of the CNN, that may be repeated multiple times, is a *convolution* layer followed by a *pooling* layer. Those two layers always go hand in hand. In what follows, we will detail how they interplay, using as an example the NLP task of sentence classification. The illustrative example we will follow can be seen in Figure 1.

Convolution layer. We can represent a given training instance (a sentence) by a real matrix $A \in \mathbb{R}^{(s,d)}$, where s is the length of the sentence (in number of words), and d is the dimension of the word embedding vectors[9]. The convolution layer applies a linear filter (a matrix of parameters) to each region matrix. Region matrices are sub-matrices of A obtained by concatenating the embedding vectors of the words found within each instantiation of a 1d window applied over the sentence. Note that if we were dealing with images, the window would be 2-dimensional. Let h be the size of the window (h is also known as the region size). There are $s-h+1$ instantiations of the window, hence $s-h+1$

[8] https://keras.io/preprocessing/sequence/

[9] The word vectors can either be initialized randomly or be pre-trained. In the latter case, the vectors can be further tuned during training ("non-static" approach) or fixed ("static" approach), see [10].

region matrices in total. The output of the convolution layer for a given filter is a vector $o \in \mathbb{R}^{s-h+1}$ whose elements are computed as:

$$o_i = W \cdot A[i : i + h - 1,] \tag{2}$$

Where $A[i : i+h-1,] \in \mathbb{R}^{(h,d)}$ is the i^{th} region matrix, $W \in \mathbb{R}^{(h,d)}$ is the parameter matrix associated with the filter (initialized randomly and learned during training), and $\cdot$ is an operator returning the sum of the row-wise dot product of the two matrices. Note that for a given filter, the same W is applied to all instantiations of the window regardless of their positions in the sentence. This is precisely what gives the spatial invariance property to the model, because the filter is trained to recognize a pattern wherever it is located.

Then, a nonlinear activation function f, such as `ReLU`[10] ($\max(0, x)$) or `tanh` ($\frac{e^{2x}-1}{e^{2x}+1}$), is applied elementwise to o, returning what is known as the *feature map* $c \in \mathbb{R}^{s-h+1}$ associated with the filter:

$$c_i = f(o_i + b_i) \tag{3}$$

Where $b \in \mathbb{R}^{s-h+1}$ is a bias vector. In practice, as shown in Figure 1, different window sizes are used, and various filters are applied to each region, in order to give the model the ability to learn different, complementary features for each region.

Pooling layer. Once the feature maps are available, there is one last layer in the network that takes them as input and outputs class probabilities via a softmax activation function. However, since the length of each feature map varies depending on the length of the sentence and the size of the window, and since neural classifiers require a fixed-size input vector, a *pooling* strategy is employed. One of the most common variants of pooling is k-max pooling, where the k greatest values of each feature map are extracted and concatenated to form a final vector whose size always remains constant during training. Pooling also serves a dimensionality reduction purpose.

A note on the softmax function. The softmax function is a nonlinearity that is applied to the output layer only (not to the hidden layers). It can transform a vector $x \in \mathbb{R}^K$ into a vector of positive floats that sum to one, i.e., into a probability distribution over the classes to be predicted.

$$\text{softmax}(x_i) = \frac{e^{x_i}}{\sum_{j=1}^{K} e^{x_j}} \tag{4}$$

In the binary classification case, you will see that sometimes instead of having a final output layer of two neurons with a softmax, where each neuron represents one of the two classes, we have an output layer with only a single neuron and a sigmoid function ($\sigma(x) = \frac{1}{1+e^{-x}}$). In that case, decision is made based on whether $\sigma(x)$ is greater or smaller than 0.5. These two approaches are equivalent. Indeed, $\frac{1}{1+e^{-x}} = \frac{e^x}{e^x+e^0}$. So, the one-neuron sigmoid layer can be viewed as a two-neuron softmax layer where one of the neurons never activates and has its output always equal to zero.

[10] compared to tanh, ReLu is affordable and can better combat the *vanishing gradients* problem as it does not saturate.

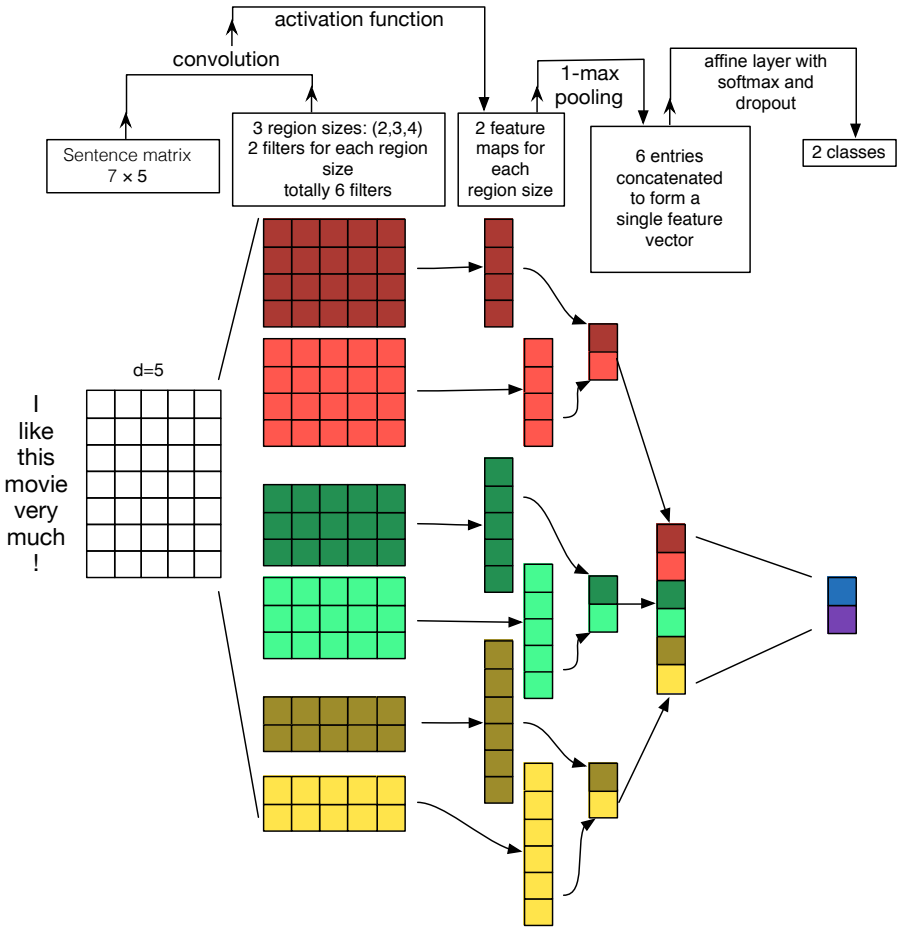

Figure 1: CNN architecture for sentence classification, taken from Zhang and Wallace (2015). $s = 7$, $d = 5$. Three regions of respective sizes $h = \{2, 3, 4\}$ are considered, with associated output vectors of resp. lengths $s - h + 1 = \{6, 5, 4\}$ for each filter (produced after convolution, not shown). There are two filters per region size. For the three region sizes, the filters are resp. associated with feature maps of lengths $\{6, 5, 4\}$ (the output vectors after elementwise application of f). 1-max pooling is used. Note that in practice, the sentence would have been pre-processed.

As illustrated by Figure 1, looking at things from a high level, the CNN architecture connects each (filtered) region of the input to a single neuron in the final softmax layer. Therefore, after training, each neuron can be viewed as "focusing" on a particular pattern. For instance, a given neuron may learn to only activate when "good","great", "awesome", "love it", etc. are observed in the input.

4.3 Coding time

We will use a CNN to predict the polarity of movie reviews as described in section 2. You can find the code in the `imdb_cnn.py` file. Go through it, and see how the different layers are added. Does that make sense to you? Remember that you can find useful information in the Keras documentation.

5 Long Short Term Memory networks (LSTMs)

Since LSTMs are a specific version of Recurrent Neural Networks (RNNs), we first present RNNs.

5.1 RNNs

While CNNs are naturally good at dealing with grids, RNNs were specifically developed to be used with *sequences* [2]. Some examples include time series, or, in NLP, words (sequences of letters) or sentences (sequences of words). For instance, the task of *language modeling* consists in learning the probability of observing the next word in a sentence given the $n-1$ preceding words, that is $P[w_n|w_1, ..., w_{n-1}]$. Character-level language models are also popular. RNNs trained with such objectives can be used to generate new and quite convincing sentences from scratch, as well demonstrated in this very interesting blogpost[11]. Note that CNNs do allow to capture some order information, but it is limited to *local* patterns. Long-range dependencies are ignored [4].

As shown in Figure 2, a RNN can be viewed as a chain of simple neural layers that *share* the same parameters.

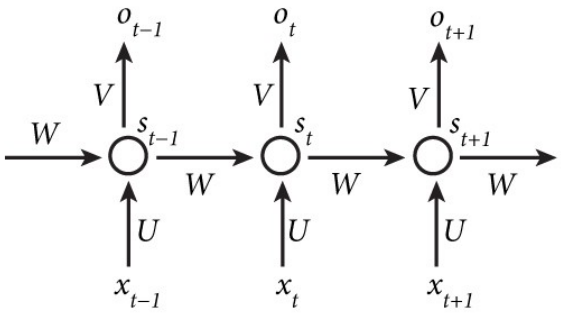

Figure 2: Three steps of an unrolled RNN (*adapted from Denny Britz' blog*)

Formally, a RNN takes as input an ordered list of input vectors $\{x_0, ..., x_T\}$ as well as an initial state vector s_{-1} (initialized to all zeroes), and returns an ordered list of state vectors $\{s_0, ..., s_T\}$, or "hidden states" (the memory of the network), as well as an ordered list of output vectors $\{o_0, ..., o_T\}$. At any given time step t (or position in the sequence, when there is no notion of time) the hidden state s_t is defined in terms of the previous hidden state s_{t-1} and the current input vector x_t (new training example) in the following *recursive* way (we omit the bias vectors for simplicity):

$$s_t = f(Ux_t + Ws_{t-1}) \tag{5}$$

Where f is a nonlinearity such as `tanh` (applied elementwise), and $U \in \mathbb{R}^{H,d_{in}}$ and $W \in \mathbb{R}^{H,H}$ are parameter matrices shared by all time steps. $x_t \in \mathbb{R}^{d_{in}}$, where d_{in} can be the size of the vocabulary, if one-hot vectors are passed as input, or the size of the embedding space, when working with shared features. Finally, $s_t \in \mathbb{R}^H$, where H is the dimension of the hidden layer, that is, the number of neurons composing that layer. The larger this layer, the greater the capacity of the memory. However, this comes at the expense of a greater computational cost. Typical values for H are of 10^2 in order of magnitude.

The output vector o_t depends on the current hidden state. In the case of a classification task, it is computed as:

$$o_t = \text{softmax}(Vs_t) \tag{6}$$

[11] http://karpathy.github.io/2015/05/21/rnn-effectiveness/

Where $V \in \mathbb{R}^{d_{out},H}$ is a parameter matrix shared across all time steps, and $o_t \in \mathbb{R}^{d_{out}}$, where d_{out} depends on the task. For instance, $d_{out} = 3$ for a three-class classification task.

RNNs can run very deep: when considering sentences for instance (sequences of words), T can approach 10 or 15. This greatly amplifies the adverse effects of the well-known *vanishing gradients* problem. Note that this issue can also be experienced with feed-forward neural networks, such as the Multi-Layer Perceptron, but it just gets worse with RNN due to their inherent tendency to be deep. We won't explain how this problem arises, but basically, the take-away point is that vanishing gradients prevent long-range dependencies to be learned. Please refer to this blogpost for instance for more information.

5.2 LSTMs

In practice, whenever people use RNNs, they use LSTMs. LSTMs have a specific architecture that allows them to escape vanishing gradients and keep track of information over long time periods [5].

As shown in Figure 3, the two things that change in LSTMs compared to basic RNNs are (1) the presence of a *cell state* (c_t) and (2) how hidden states are computed. With vanilla RNNs, we have a single layer where $s_t = \tanh(Ux_t + Ws_{t-1})$. With LSTMs, there are four layers that interplay to form a gating mechanism which removes or adds information from/to the cell state. This gives the network the ability to remember or forget specific information about the preceding elements as it is being fed the sequence of training examples.

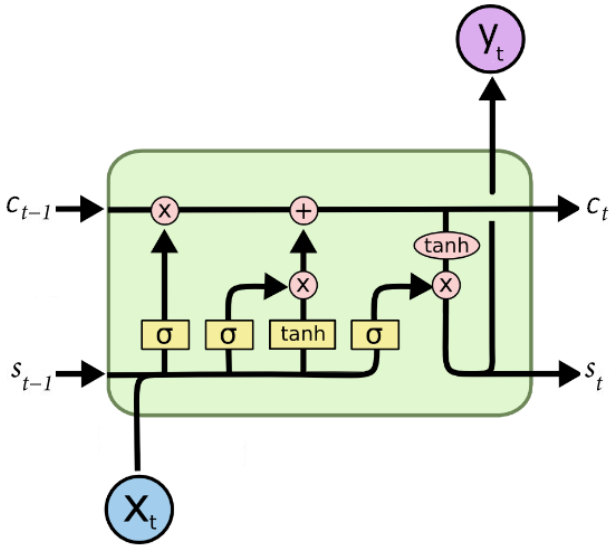

Figure 3: One element of a LSTM chain. Adapted from *Chris Colah's blog*. Note that to avoid confusion with the notation used for the output gate layer, the output vector (o_t in Figure 2) has been renamed y_t

More specifically, the four layers include:

1. forget gate layer: $f_t = \sigma\big(U_f x_t + W_f s_{t-1}\big)$
2. input gate layer: $i_t = \sigma\big(U_i x_t + W_i s_{t-1}\big)$
3. candidate values computation layer: $\tilde{c}_t = \tanh\big(U_c x_t + W_c s_{t-1}\big)$
4. output gate layer: $o_t = \sigma\big(U_o x_t + W_o s_{t-1}\big)$

Thanks to the elementwise application of the sigmoid function (σ), the *gate* layers 1, 2, and 4 generate vectors whose entries are all comprised between zero and one. When one of these layers is multiplied with another vector, it thus acts as a filter that only selects a certain proportion of that vector. This is precisely why those layers are called gates. The two extreme cases are when all entries are equal to 1 (the full vector passes) or to 0 (nothing passes). Note that the three gates are computed in the exact same way, only the parameters (shared by all time steps) vary.

By taking into account the new training example x_t and the current memory s_{t-1}, the forget gate layer f_t determines how much of the previous cell state c_{t-1} should be forgotten, while from the same information, the input gate layer i_t decides how much of the candidate values $\tilde{c}_t$ should be added to the cell state, or in other words, how much of the new information should be learned:

$$c_t = f_t * c_{t-1} + i_t * \tilde{c}_t \tag{7}$$

Where $*$ denotes elementwise multiplication. Finally, the proportion of the updated cell state that we want to expose to the next time steps is controlled by the output gate layer o_t:

$$s_t = \tanh\big(c_t\big) * o_t \tag{8}$$

And, as before in the simple RNN, the output vector is computed as a function of the new hidden state. For instance, in the case of a multi-class classification task, we would have:

$$y_t = \text{softmax}(V s_t) \tag{9}$$

Analogy with vanilla RNN. If we decide to forget everything about the previous state (all elements of f_t are null), to learn all of the new information (all elements of i_t are equal to 1), and to memorize the entire cell state to pass to the next time step (all elements of o_t are equal to 1), we have $c_t = \tilde{c}_t = \tanh\big(U_c x_t + W_c s_{t-1}\big)$, and thus we go back to a vanilla RNN, the only difference being the additional `tanh` (we end up with $s_t = \tanh\big(\tanh\big(U_c x_t + W_c s_{t-1}\big)\big)$ instead of $s_t = \tanh\big(U_c x_t + W_c s_{t-1}\big)$ like with the classical RNN).

Variations. Some variants of LSTMs are very popular. Although we won't deal with them today, it is worth knowing about the LSTM with "peepholes" [3], or more recently the Gated Recurrent Unit or GRU [1]. The latter proposes a simpler architecture, with only two gates, and where the cell state is merged with the hidden state (there is no c_t).

5.3 Coding time

We will use a LSTM to predict the polarity of movie reviews as described in section 2. You can find the code in the `imdb_lstm.py` file. Go through it and see how the different layers are added. Remember that useful information can be found in Keras' documentation. Is the architecture consistent with the brief theoretical introduction we previously provided? Try to tweak some of the parameters. Especially, some important parameters are the optimizer,

the batch size, and the number of epochs. You can also compare the performance of the standard LSTM against that of the Gated Recurrent Unit (GRU), and the vanilla RNN. Finally, how is performance compared to the CNN?

6 Going further

CNNs (for image processing) and LSTMs (for natural language generation) have been successfully combined using "encoder-decoder" architectures in image and video captioning tasks [12]. Some publicly available tools allow you to upload an image and get the caption for it[12].

References

[1] Cho, K., Van Merriënboer, B., Gulcehre, C., Bahdanau, D., Bougares, F., Schwenk, H., Bengio, Y. (2014). Learning phrase representations using RNN encoder-decoder for statistical machine translation. arXiv preprint arXiv:1406.1078.

[2] Elman, J. L. (1990). Finding structure in time. Cognitive Science, 14:2, 179-211.

[3] Gers, Felix A., and Jürgen Schmidhuber. Recurrent nets that time and count. Neural Networks, 2000. IJCNN 2000, Proceedings of the IEEE-INNS-ENNS International Joint Conference on. Vol. 3. IEEE, 2000.

[4] Goldberg, Y. (2015). A primer on neural network models for natural language processing. Journal of Artificial Intelligence Research, 57, 345-420.

[5] Hochreiter, S., Schmidhuber, J. (1997). Long short-term memory. Neural computation, 9(8), 1735

[6] Johnson, R., Zhang, T. (2015). Effective Use of Word Order for Text Categorization with Convolutional Neural Networks. To Appear: NAACL-2015, (2011).

[7] Johnson, R., Zhang, T. (2015). Semi-supervised Convolutional Neural Networks for Text Categorization via Region Embedding.

[8] Kaiming He and Xiangyu Zhang and Shaoqing Ren and Jian Sun. Deep Residual Learning for Image Recognition. 2016. CVPR

[9] Kalchbrenner, N., Grefenstette, E., Blunsom, P. (2014). A Convolutional Neural Network for Modelling Sentences. Acl, 655–665.

[10] Kim, Y. (2014). Convolutional Neural Networks for Sentence Classification. Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing (EMNLP 2014), 1746–1751.

[11] LeCun, Y., Bottou, L., Bengio, Y., and Haffner, P. (1998). Gradient-based learning applied to document recognition. Proceedings of the IEEE, 86(11), 2278-2324.

[12] Oriol Vinyals, Alexander Toshev, Samy Bengio, Dumitru Erhan (2015). Show and Tell: A Neural Image Caption Generator. CVPR

[12] https://github.com/apple2373/chainer-caption

[13] Santos, C. N. dos, Gatti, M. (2014). Deep Convolutional Neural Networks for Sentiment Analysis of Short Texts. In COLING-2014 (pp. 69–78).

[14] Zhang, Y., and Wallace, B. (2015). A sensitivity analysis of (and practitioners' guide to) convolutional neural networks for sentence classification. arXiv preprint arXiv:1510.03820.

[15] Zhang, X., Zhao, J., LeCun, Y. (2015). Character-level Convolutional Networks for Text Classification, 1–9.