Introduction to CNNs and LSTMs for NLP

Antoine J.-P. Tixier*
DaSciM team, École Polytechnique

March 23, 2017 - last updated November 29, 2017

1 Disclaimer

I put together these notes as part of my TA work for the *Graph and Text Mining* grad course of Prof. Michalis Vazirgiannis in the Spring of 2017. They accompanied a programming lab session about Convolutional Neural Networks (CNNs) and Long Short Term Memory networks (LSTMs) for document classification, using Python and Keras[1]. Keras is a very popular Python library for deep learning. It is a wrapper for `TensorFlow`, `Theano`, and `CNTK`.

To write this handout, I curated information mainly from: the original 2D CNN paper [11] and Stanford's `CS231n` CNN course notes, Zhang and Wallace practitioners' guide to CNNs in NLP [18], the seminal papers on CNN for text classification [8, 9]; Denny Britz' tutorial on RNNs, and Chris Colah's post on understanding LSTMs. Last but not least, Yoav Golderg's primer on neural networks for NLP [5] proved very useful in understanding both CNNs and RNNs.
The CNN part of the code can be found on my GitHub here.

2 IMDB Movie review dataset

We will be performing binary classification (positive/negative) on reviews from the Internet Movie Database (IMDB) dataset[2]. This task is known as *sentiment analysis* or *opinion mining*. The IMDB dataset contains 50K movie reviews, labeled by polarity (pos/neg). The data are partitioned into 50 % for training and 50% for testing. We could access the dataset directly through the `imdb.load_data()` Keras method, but to have more control over preprocessing and for learning purposes, we start from scratch with the raw data. The `imdb_preprocess.py` cleans the reviews and put them in a format suitable to be passed to neural networks: each review is a list of word indexes (integers) from a dictionary of size V where the most frequent word has index 1.

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

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

*antoine.tixier-1@colorado.edu
[1] https://keras.io/getting-started/faq/
[2] http://ai.stanford.edu/~amaas/data/sentiment/
[3] https://keras.io/objectives/

Where N is the number of observations, p_i is the probability assigned to class 1, $(1 - p_i)$ is the probability assigned to class 0, and y_i is the true label of the ith 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 $-\log(0.8) = 0.22$. 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 will reach 2.3. Thus, the further away 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 [5]. The embeddings are initialized randomly or obtained from pre-training[4]. They can then be updated during training just like other model parameters, or be kept static.

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. With traditional one-hot vectors, those two features would be considered orthogonal and predictive power would not be able to be shared between them[5]. Also, going from a huge sparse space to a dense and compact space reduces computational cost and the amount of data required to fit the model, since there are fewer parameters to learn.

3.3 Combining core features

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 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 *truncation*[6] need to be used so that the same number of vectors are concatenated for each training example. For instance, regardless of its size, every document in the collection can be transformed to have the same fixed length s: the longer documents are truncated to their first (or last, middle...) s words, and the shorter documents are padded with a special zero vector to make up for the missing words [18, 9].

[4] In NLP, pre-trained word vectors obtained with Word2vec or GloVe from very large corpora are often used. E.g., Google News `word2vec` vectors can be obtained from https://code.google.com/archive/p/word2vec/, under the section “Pre-trained word and phrase vectors”

[5] 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

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

4 Convolutional Neural Networks (CNNs)

4.1 Local invariance and compositionality

Initially inspired by studies of the cat's visual cortex [7], CNNs were developed in computer vision to work on regular grids such as images [11]. They are feedforward neural networks where each neuron in a layer receives input from a neighborhood of the neurons in the previous layer. Those neighborhoods, or *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. This property is called *compositionality*. 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 absolute positions of the features in the image do not matter. Only capturing their respective positions is useful for composing higher-level patterns. So, the model should be able to detect a feature regardless of its position in the image. This property is called *local invariance*. Compositionality and local invariance are the two key concepts of CNNs.

CNNs have reached very good performance in computer vision [10], but it is not difficult to understand that thanks to compositionality and local invariance, they can also do very well in NLP. Indeed, in NLP, high-order features (n-grams) can be constructed from lower-order features just like in CV, and ordering is crucial locally ("not bad, quite good", "not good, quite bad", "do not recommend"), but not at the document level. 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 document. 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 preferred.

4.2 Convolution and pooling

Though recent work suggests that convolutional layers may directly be stacked on top of each other [15], the elementary construct of the CNN is a *convolution* layer followed by a *pooling* layer. In what follows, we will detail how these two layers interplay, using as an example the NLP task of short document classification (see Figure 1).

Input. We can represent a document as a real matrix $A \in \mathbb{R}^{s \times d}$, where s is the document length, and d is the dimension of the word embedding vectors. Since s is fixed at the collection level but the documents are of different sizes, we truncate the longer documents to their first s words, and pad the shorter documents with a special zero vector. The word vectors may either be initialized randomly or be pre-trained, and can be further tuned during training or not ("non-static" vs. "static" approach [9]).

Thinking of A as an image is misleading, because there is only one spatial dimension. The embedding vectors are not actually part of the input itself, they just represent the coordinates of the elements of the input in a shared latent space. In computer vision, the term *channels* is often used to refer to this *depth* dimension (not to be mistaken with the number of hidden layers in the network). If we were dealing with images, we would have two spatial dimensions, plus the depth. The input would be a tensor of dimensionality (width × height × n_channels), i.e., a 2D matrix where each entry would be associated with a vector of length 3 or 1, respectively in the case of color (RGB) and grey level images.

Convolution layer. The convolution layer is a linear operation followed by a nonlinear transformation. The linear operation consists in multiplying (elementwise) each instantiation of a 1D window applied over the input document by a *filter*, represented as a matrix of parameters. The filter, just like the window, has only one spatial dimension, but it extends fully through

the input depth (the d dimensions of the word embedding space). If h is the window size, the parameter matrix W associated with the filter thus belongs to $\mathbb{R}^{h \times d}$. W is initialized randomly and learned during training.

The instantiations of the window over the input are called *regions* or *receptive fields*. There are $^{(s-h)}/_{\text{stride}} + 1$ of them, where stride corresponds to the number of words by which we slide the window at each step. With a stride of 1, there are therefore $s - h + 1$ receptive fields. The output of the convolution layer for a given filter is thus 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 $A[i : i + h - 1, :] \in \mathbb{R}^{h \times d}$ is the ith region matrix, , and $\cdot$ is an operator returning the sum of the row-wise dot product of 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 document. In other words, the parameters of the filter are shared across receptive fields. 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. It also greatly reduces the total number of parameters of the model.

Then, a nonlinear activation function f, such as `ReLU`[7] ($\max(0, x)$) 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:

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

Where $b \in \mathbb{R}$ is a trainable bias.

For short sentence classification, best region sizes are generally found between 1 and 10, and in practice, n_f filters (with $n_f \in [100, 600]$) are applied to each region to give the model the ability to learn different, complementary features for each region [18]. Since each filter generates a feature map, each region is thus embedded into an n_f-dimensional space. Moreover, using regions of varying size around the optimal one improves performance [18]. In that case, different parallel branches are created (one for each region size), and the outputs are concatenated after pooling, as shown in Figure 1. Performance and cost increase with n_f up to a certain point, after which the model starts overfitting.

Pooling layer. The exact positions of the features in the input document do not matter. What matters is only whether certain features are present or absent. For instance, to classify a review as positive, whether "best movie ever" appears at the beginning or at the end of the document is not important. To inject such robustness into the model, *global k-max pooling*[8] is employed. This approach extracts the k greatest values from each feature map and concatenates them, thus forming a final vector whose size always remains constant during training. For short sentence classification, [18] found that $k = 1$ was by far superior to higher-order strategies. They also reported that using the maximum was much better than using the average, which makes sense, since we're only interested in extracting the most salient feature from each feature map.

[7] compared to tanh, ReLu is affordable (sparsity induced by many zero values in the negative regime) and better combats the *vanishing gradients* problem as in the positive regime, the gradient is constant, whereas with tanh it becomes increasingly small

[8] pooling may also be applied locally over small regions, but for short text classification, global pooling works better [18].

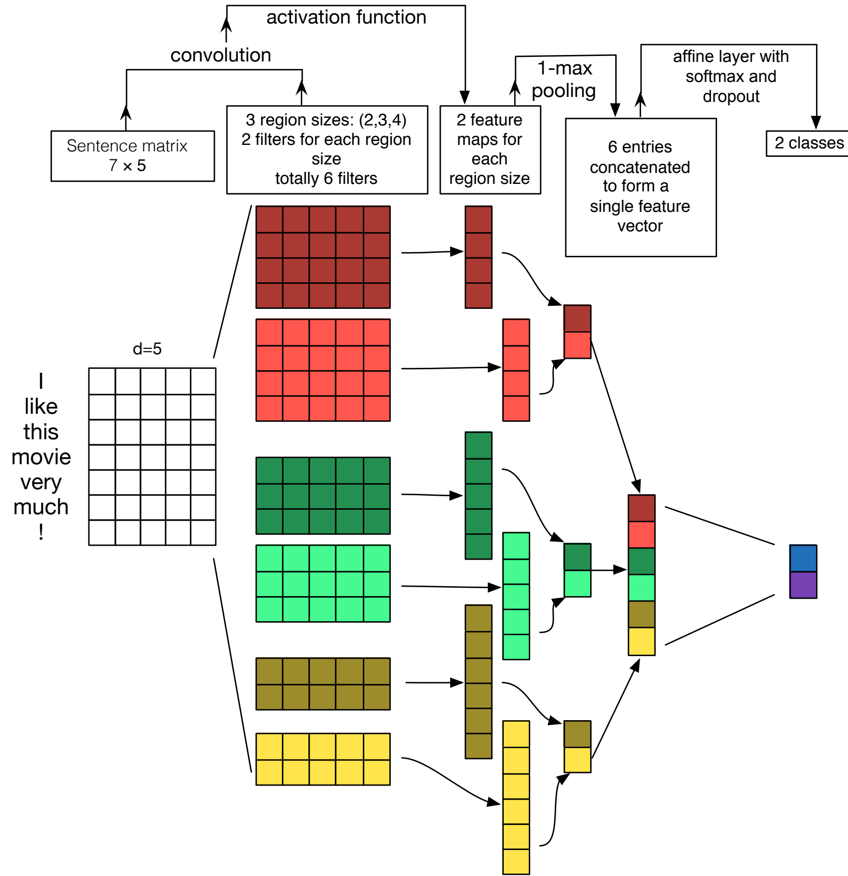

Figure 1: CNN architecture for (short) document classification, taken from Zhang and Wallace (2015) [18]. $s = 7$, $d = 5$. 3 regions of respective sizes $h = \{2, 3, 4\}$ are considered, with associated output vectors of resp. lengths $s - h + 1 = \{6, 5, 4\}$ for each filter (produced after convolution, not shown). There are 2 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 and addition of bias). 1-max pooling is used.

Document encoding. As shown in Figure 1, looking at things from a high level, the CNN architecture connects each filtered version of the input to a single neuron in a final feature vector. This vector can be seen as an embedding, or *encoding*, of the input document. It is the main contribution of the model, the thing we're interested in. The rest of the architecture just depends on the task.

Softmax layer. Since the goal here is to classify documents, a softmax function is applied to the document encoding to output class probabilities. However, different tasks would call for different architectures: determining whether two sentences are paraphrases, for instance, would require two CNN encoders sharing weights, with a final energy function and a contrastive loss (à la Siamese [2]); for translation or summarization, we could use a LSTM language model decoder conditioned on the CNN encoding of the input document (à la `seq-to-seq` [17]), etc.

Going back to our classification setting, the softmax transforms 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, instead of having a final output layer of two neurons with a softmax, where each neuron represents one of the two classes, we can have an output layer with only one neuron and a sigmoid function ($\sigma(x) = \frac{1}{1+e^{-x}}$). In that case, the neuron outputs the

probability of belonging to one of the two classes, and decision regarding the class to predict is made based on whether $\sigma(x)$ is greater or smaller than 0.5 (assuming equal priors). 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.

4.3 Number of parameters

The total number of trainable parameters for our CNN is the sum of the following terms:

- **word embedding matrix** (only if non-static mode): $(V+1) \times d$, where V is the size of the vocabulary. We add one row for the zero-padding vector.
- **convolution layer**: $h \times d \times n_f + n_f$ (the number of entries in each filter by the number of filters, plus the biases).
- **softmax layer**: $n_f \times 1 + 1$ (fully connected layer with an output dimension of 1 and one bias).

4.4 Visualizing and understanding inner representations and predictions

Document embeddings. A fast and easy way to verify that our model is learning effectively is to check whether its internal document representations make sense. Recall that the feature vector which is fed to the softmax layer can be seen as an n_f-dimensional encoding of the input document. By collecting the intermediate output of the model at this precise level in the architecture for a subset of documents, and projecting the vectors to a low-dimensional map, we can thus visualize whether there is any correlation between the embeddings and the labels. Figures 2 and 3 prove that indeed, our model is learning meaningful representations of documents.

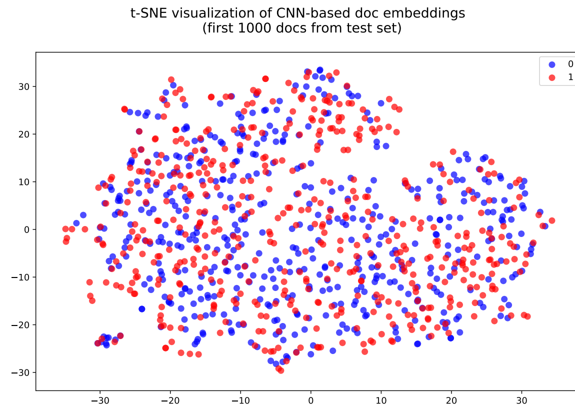

Figure 2: Doc embeddings before training.

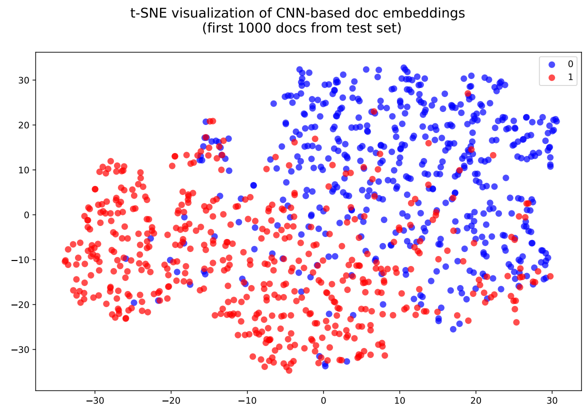

Figure 3: Doc embeddings after 2 epochs.

Predictive regions identification. This approach is presented in section 3.6 (Tables 5 & 6) of [8]. Recall that before we lose positional information by applying pooling, each of the n_f filters of size h is associated with a vector of size $(s-h)/\text{stride} + 1$ (a feature map) whose entries represent the output of the convolution of the filter with the corresponding receptive field in the input, after application of the nonlinearity and addition of the bias. Therefore, each receptive field is embedded into an n_f-dimensional space. Thus, after training, we can identify the regions of a given document that are the most predictive of its category by inspecting the intermediate output of the model corresponding to the receptive field embeddings (right before the pooling layer), and by finding the regions that have the highest norms. For instance, some of the most

predictive regions for negative IMDB reviews are: "worst movie ever", "don't waste your money", "poorly written and acted", "awful picture quality". Conversely, some regions very indicative of positivity are: "amazing soundtrack", "visually beautiful", "cool journey", "ending quite satisfying"...

Saliency maps. Another way to understand how the model is issuing its predictions was described by [14] and applied to NLP by [12]. The idea is to rank the elements of the input document $A \in \mathbb{R}^{s \times d}$ based on their influence on the prediction. An approximation can be given by the magnitudes of the first-order partial derivatives of the output of the model $\text{CNN} : A \mapsto \text{CNN}(A)$ with respect to each row a of A:

$$\text{saliency}(a) = \left| \frac{\partial(\text{CNN})}{\partial a} |_a \right| \tag{5}$$

The interpretation is that we identify which words in A need to be *changed the least to change the class score the most*. The derivatives can be obtained by performing a single back-propagation pass (based on the prediction, not the loss like during training). Figures 4 and 5 show saliency map examples for negative and positive reviews, respectively.

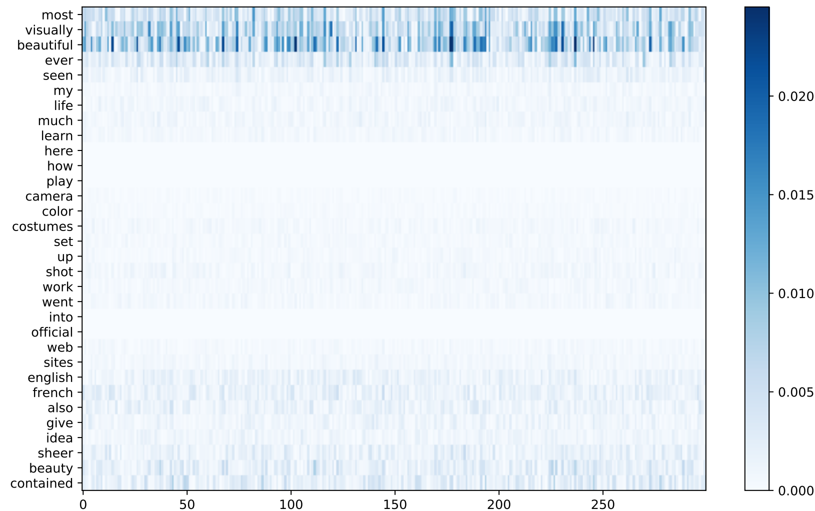

Figure 4: Saliency map for document 1 of the IMDB test set (true label: positive)

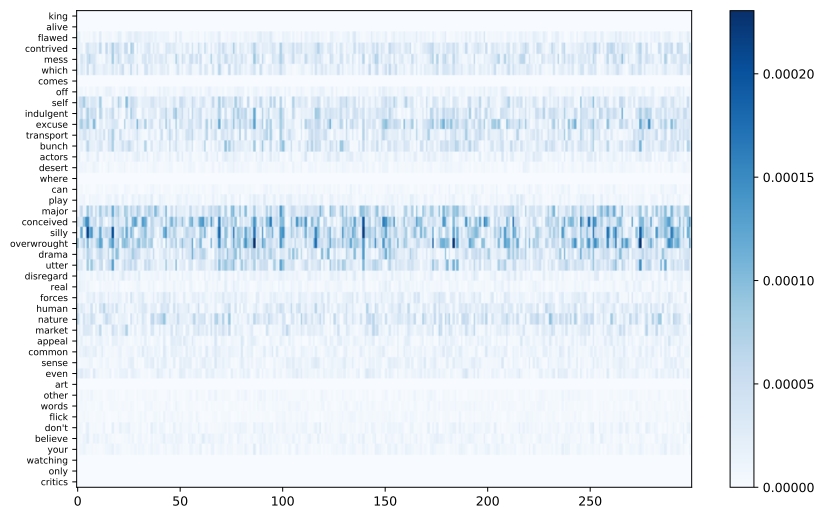

Figure 5: Saliency map for document 15 of the IMDB test set (true label: negative)

5 Long Short Term Memory networks (LSTMs)

Since LSTMs are a specific version of Recurrent Neural Networks (RNNs), we first present RNNs. A good review of RNNs, LSTMs and their applications can be found in [13].

5.1 RNNs

While CNNs are naturally good at dealing with grids, RNNs were specifically developed to be used with *sequences* [3]. Some examples include time series, or, in NLP, words (sequences of characters) 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[9]. CNNs do allow to capture some order information, but it is limited to *local* patterns, and long-range dependencies are ignored [5]. As shown in Figure 6, a RNN can be viewed as a chain of simple neural layers that *share* the same parameters.

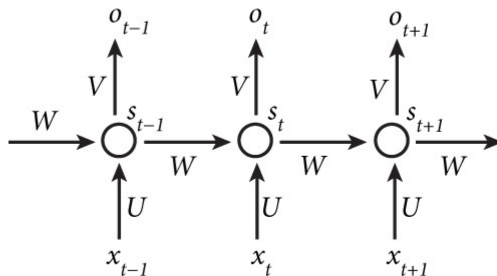

Figure 6: 3 steps of an unrolled RNN (*adapted from Denny Britz' blog*)

From a high level, a RNN is fed 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 $\{o_0, ..., o_T\}$. Note that each training example is a $\{x_0, ..., x_T\}$ sequence of its own.

At any position t in the sequence, the hidden state s_t is defined in terms of the previous hidden state s_{t-1} and the current input vector x_t in the following *recursive* way:

$$s_t = f(Ux_t + Ws_{t-1} + b) \tag{6}$$

Where f is a nonlinearity such as `tanh` (applied elementwise), $x_t \in \mathbb{R}^{d_{in}}$, and $U \in \mathbb{R}^{H \times d_{in}}$, $W \in \mathbb{R}^{H \times H}$ are parameter matrices shared by all time steps, and s_t, s_{t-1}, and b belong to $\mathbb{R}^H$. d_{in} can be the size of the vocabulary, if one-hot vectors are passed as input, or the dimensionality of the embedding space, when working with shared features. $H \sim 100$ is the dimension of the hidden layer. The larger this layer, the greater the capacity of the memory (with an increase in computational cost).

The output vector $o_t \in \mathbb{R}^{d_{out}}$ depends on the current hidden state $s_t \in \mathbb{R}^H$. For classification, it is computed as:

$$o_t = \text{softmax}(Vs_t) \tag{7}$$

Where $V \in \mathbb{R}^{d_{out} \times H}$ is a parameter matrix shared across all time steps. d_{out} depends on the task (e.g, $d_{out} = 3$ for 3-class classification).

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

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* and *exploding* 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 depth. We won't explain how this problem arises, but the take-away point is that vanishing/exploding gradients prevent long-range dependencies from being learned. You may 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/exploding gradients and keep track of information over longer time periods [6].

As shown in Figure 7, 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 $s_t = \tanh(Ux_t + Ws_{t-1} + b)$. With LSTMs, there are not one but 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.

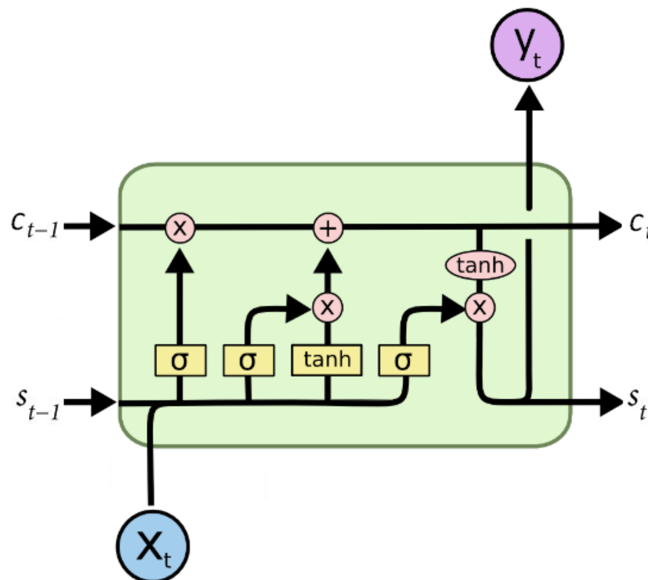

Figure 7: One element of a LSTM chain. Adapted from *Chris Colah's blog*. Note that to avoid confusion with the notation used for the output gate layer, the output vector (previously o_t in Figure 6) has been renamed y_t.

More specifically, the four layers include:

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

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 0 and 1. 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 3 gates are computed in the exact same way, only the parameters (shared by all time steps) vary.

Forgetting/learning. 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:

$$c_t = f_t * c_{t-1} + i_t * \tilde{c}_t \tag{8}$$

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:

$$s_t = \tanh\big(c_t\big) * o_t \tag{9}$$

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:

$$y_t = \text{softmax}(V s_t) \tag{10}$$

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 $c_t = \tilde{c}_t = \tanh\big(U_c x_t + W_c s_{t-1} + b_c\big)$, and thus we go back to a vanilla RNN, the only difference being an additional `tanh`, as we end up with $s_t = \tanh\big(\tanh\big(U_c x_t + W_c s_{t-1} + b_c\big)\big)$ instead of $s_t = \tanh\big(U_c x_t + W_c s_{t-1} + b_c\big)$ like in the classical RNN case.

Variations. Some variants of LSTMs are very popular: it is worth learning about the LSTM with “peepholes” [4], and 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).

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, e.g., [16].

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] Chopra, Sumit, Raia Hadsell, and Yann LeCun. "Learning a similarity metric discriminatively, with application to face verification." Computer Vision and Pattern Recognition, 2005. CVPR 2005. IEEE Computer Society Conference on. Vol. 1. IEEE, 2005.

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

[4] Gers, F. A., and Schmidhuber, J.. 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.

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

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

[7] Hubel, David H., and Torsten N. Wiesel (1962). Receptive fields, binocular interaction and functional architecture in the cat's visual cortex. The Journal of physiology 160.1:106-154.

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

[9] 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.

[10] Krizhevsky, Alex, Ilya Sutskever, and Geoffrey E. Hinton. "Imagenet classification with deep convolutional neural networks." Advances in neural information processing systems. 2012.

[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] Li, J., Chen, X., Hovy, E., and Jurafsky, D. (2015). Visualizing and understanding neural models in nlp. arXiv preprint arXiv:1506.01066.

[13] Lipton, Zachary C., John Berkowitz, and Charles Elkan. "A critical review of recurrent neural networks for sequence learning." arXiv preprint arXiv:1506.00019 (2015).

[14] Simonyan, K., Vedaldi, A., and Zisserman, A. (2013). Deep inside convolutional networks: Visualising image classification models and saliency maps. arXiv preprint arXiv:1312.6034.

[15] Springenberg, J. T., Dosovitskiy, A., Brox, T., and Riedmiller, M. (2014). Striving for simplicity: The all convolutional net. arXiv preprint arXiv:1412.6806.

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

[17] Sutskever, Ilya, Oriol Vinyals, and Quoc V. Le. "Sequence to sequence learning with neural networks." Advances in neural information processing systems. 2014.

[18] 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.