Notes on Deep Learning for NLP

Antoine J.-P. Tixier
Computer Science Department (DaSciM team)
École Polytechnique, Palaiseau, France
antoine.tixier-1@colorado.edu

Last updated Thursday 30th August, 2018 (first uploaded March 23, 2017)

Contents

1 Disclaimer

Writing these notes is part of my learning process, so it is a work in progress. To write the current version of this document, I curated information mainly from the original 2D CNN paper [16] and Stanford's `CS231n` CNN course notes[1], Zhang and Wallace practitioners' guide to CNNs in NLP [26], the seminal papers on CNN for text classification [13, 14], Denny Britz' tutorial[2] on RNNs, Chris Colah's post[3] on understanding the LSTM unit, and the seminal papers on the GRU unit [2, 4], encoder-decoder architectures [2, 24] and attention [20, 1]. Last but not least, Yoav Golderg's primer on neural networks for NLP [8] and Luong, Cho and Manning tutorial on neural machine translation[4] proved very useful.

2 Code

I implemented some of the models described in this document in Keras and tested them on the IMDB movie review dataset. The code can be found on my GitHub: `https://github.com/Tixierae/deep_learning_NLP`. Again, this is a work in progress.

3 IMDB Movie review dataset

3.1 Overview

The task is to perform binary classification (positive/negative) on reviews from the Internet Movie Database (IMDB) dataset[5], which is known as *sentiment analysis* or *opinion mining*. The dataset contains 50K movie reviews, labeled by polarity. The data are partitioned into 50 % for training and 50% for testing. The `imdb_preprocess.py` script on my GitHub 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.

3.2 Binary classification objective function

The objective function that our models will learn to *minimize* is the *log loss*, also known as the *cross entropy*. More precisely, in a binary classification setting with 2 classes (say 0 and 1) the log loss is defined as:

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

[1] `http://cs231n.github.io/convolutional-networks/`
[2] `http://www.wildml.com/2015/09/recurrent-neural-networks-tutorial-part-1-introduction-to-rnns/`
[3] `http://colah.github.io/posts/2015-08-Understanding-LSTMs/`
[4] `https://sites.google.com/site/acl16nmt/home`
[5] `http://ai.stanford.edu/~amaas/data/sentiment/`

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 $-\log(0.8) = 0.22$. 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.

4 Paradigm switch

4.1 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 [8]. The embeddings are initialized randomly or obtained from pre-training[6]. They can then be updated during training just like other model parameters, or be kept static.

4.2 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[7]. 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.

4.3 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*[8] 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 [26, 14].

[6] 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"

[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

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

5 Convolutional Neural Networks (CNNs)

5.1 Local invariance and compositionality

Initially inspired by studies of the cat's visual cortex [12], CNNs were developed in computer vision to work on regular grids such as images [16]. 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 [15], 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.

5.2 Convolution and pooling

Though recent work suggests that convolutional layers may directly be stacked on top of each other [23], 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 Fig. 1).

5.2.1 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 must be 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 as many times as necessary. The word vectors may either be initialized randomly or be pre-trained. In the latter case, they can be updated during training or not ("non-static" vs. "static" approach [14]).

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.

5.2.2 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 i^{th} 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`[9] ($\max(0, x)$) 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 [26]. 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 [26]. In that case, different parallel branches are created (one for each region size), and the outputs are concatenated after pooling, as shown in Fig. 1. Performance and cost increase with n_f up to a certain point, after which the model starts overfitting.

5.2.3 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*[10] 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

[9] 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

[10] pooling may also be applied locally over small regions, but for short text classification, global pooling works better [26].

classification, [26] 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.

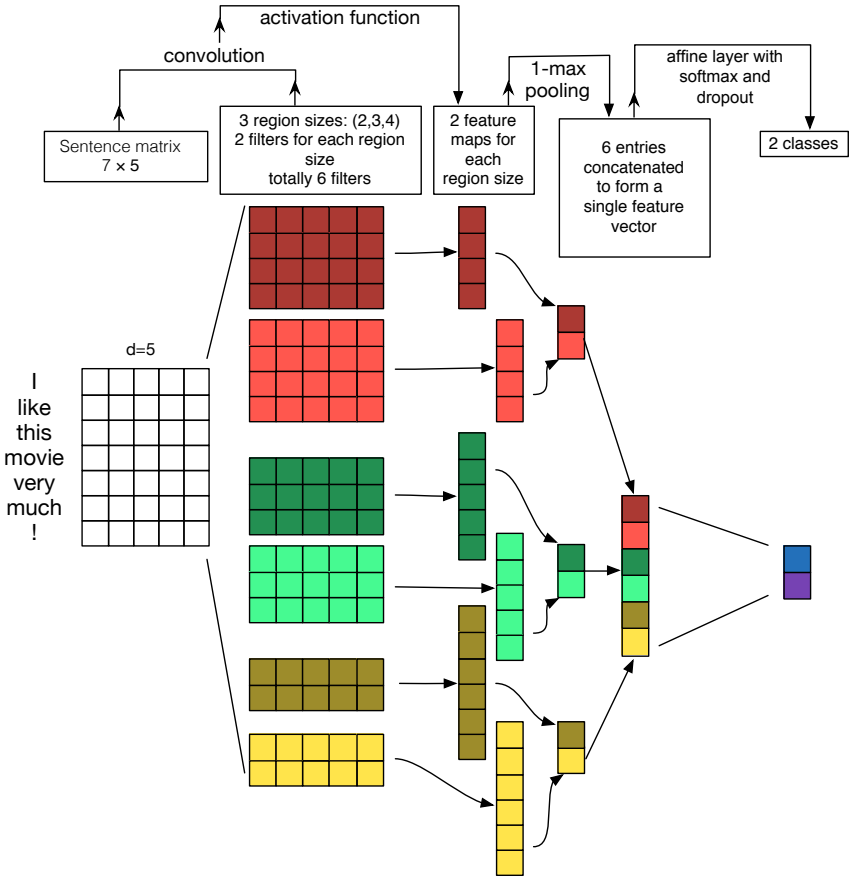

Figure 1: CNN architecture for (short) document classification, taken from Zhang and Wallace (2015) [26]. $s = 7$, $d = 5$. 3 regions of respective sizes $h = \{2, 3, 4\}$ are considered, with associated output vectors of resp. lengths $s-h+1 = \{6, 5, 4\}$ for each filter (produced after convolution, 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.

5.2.4 Document encoding

As shown in Fig. 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.

5.2.5 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 [3]); 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` [24]), 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.

5.3 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).

5.4 Visualizing and understanding inner representations and predictions

5.4.1 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. Fig.s 2 and 3 prove that indeed, our model is learning meaningful representations of documents.

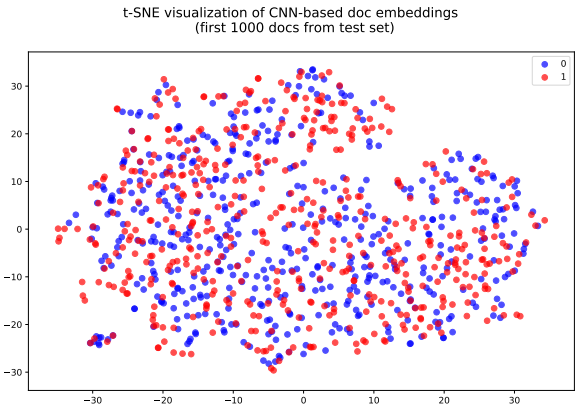

Figure 2: Doc embeddings before training.

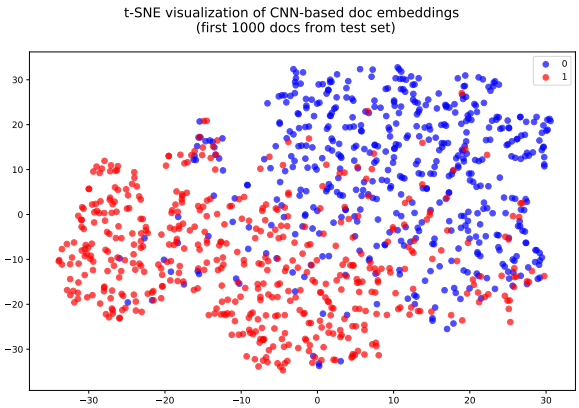

Figure 3: Doc embeddings after 2 epochs.

5.4.2 Predictive regions identification

This approach is presented in section 3.6 (Tables 5 & 6) of [13]. 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"...

5.4.3 Saliency maps

Another way to understand how the model is issuing its predictions was described by [22] and applied to NLP by [17]. 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 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). Fig.s 4 and 5 show saliency map examples for negative and positive reviews, respectively.

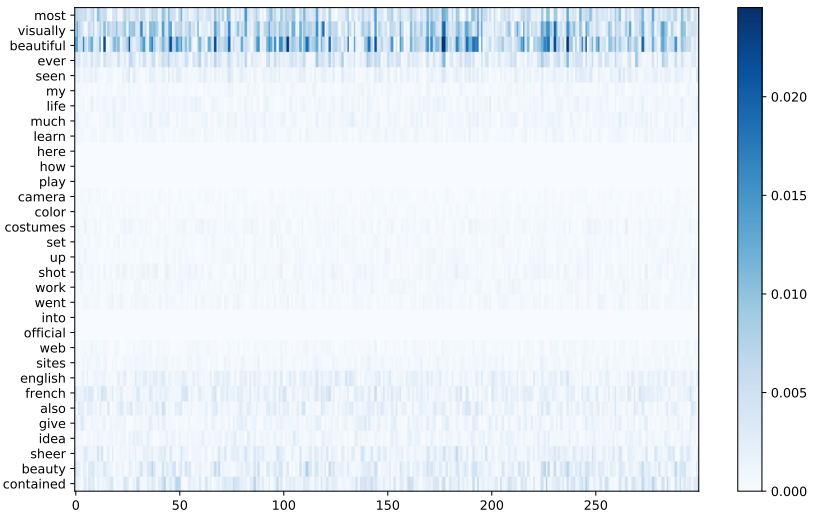

Figure 4: Saliency map for document 1 of the IMDB test set (true label: positive)

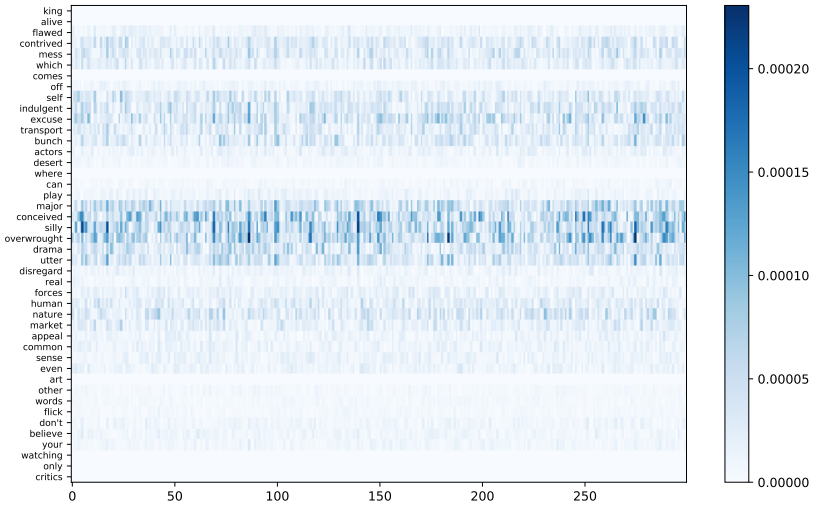

Figure 5: Saliency map for document 15 of the IMDB test set (true label: negative)

6 Recurrent Neural Networks (RNNs)

We first present the overall RNN framework, and then two types of units widely used in practice: the LSTM and the GRU. A good review of RNNs, LSTMs and their applications can be found in [19].

6.1 RNN framework

While CNNs are naturally good at dealing with grids, RNNs were specifically developed to be used with *sequences* [6]. Some examples include time series, or, in NLP, words (sequences of characters) or sentences (sequences of words). CNNs do allow to capture some order information, but it is limited to *local* patterns, and long-range dependencies are ignored [8]. As shown in Fig. 6, a RNN can be viewed as a chain of simple neural layers that *share* the same parameters.

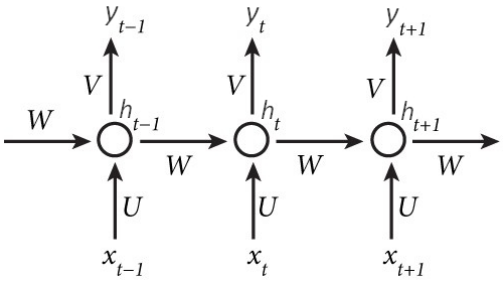

Figure 6: 3 steps of an unrolled RNN (*adapted from Denny Britz' blog*. Each circle represents a RNN unit (see equations 6 & 7).

From a high level, a RNN is fed an ordered list of input vectors $\{x_1, ..., x_T\}$ as well as an initial hidden state h_0 initialized to all zeros, and returns an ordered list of hidden states $\{h_1, ..., h_T\}$, as well as an ordered list of output vectors $\{y_1, ..., y_T\}$. The output vectors may serve as input for other RNN units, when considering deep architectures (multiple RNN layers stacked vertically, as shown in Fig. 7). The hidden states correspond more or less to the "short-term" memory of the network. Note that each training example is a full $\{x_1, ..., x_T\}$ sequence of its own, and may be associated with a label depending on the task. E.g., for short document classification, the

sequences would be associated with a label, whereas for language modeling, we would just parse all sequences, repeatedly predicting the next words.

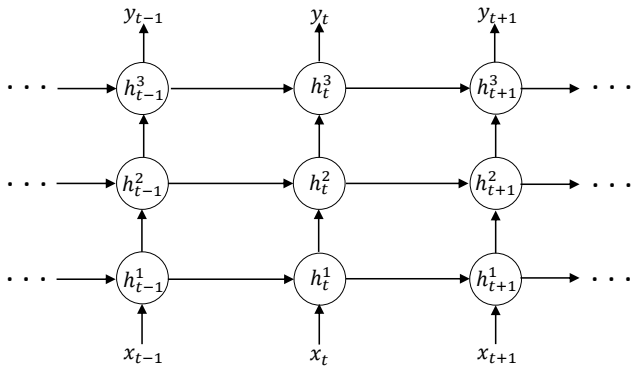

Figure 7: 3 steps of an unrolled deep RNN. Each circle represents a RNN unit. The hidden state of each unit in the inner layers (1 & 2) serves as input to the corresponding unit in the layer above.

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

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

Where f is a nonlinearity such as `tanh` (applied elementwise), $x_t \in \mathbb{R}^{d_{in}}$, $U \in \mathbb{R}^{H \times d_{in}}$ and $W \in \mathbb{R}^{H \times H}$ are parameter matrices shared by all time steps, and h_t, h_{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 is the dimension of the hidden layer. Usually, $H \sim 100$. The larger this layer, the greater the capacity of the memory, with an increase in computational cost.

The output vector $y_t \in \mathbb{R}^{d_{out}}$ transforms the current hidden state $h_t \in \mathbb{R}^H$ in a way that depends on the final task. For classification, it is computed as:

$$y_t = \text{softmax}(Vh_t) \tag{7}$$

Where $V \in \mathbb{R}^{d_{out} \times H}$ is a parameter matrix shared across all time steps. d_{out} depends on the number of categories. E.g., for 3-class document classification, $d_{out} = 3$, for a word-level language model, $d_{out} = |V|$.

Note that when stacking multiple RNN layers vertically (deep RNN architecture), the hidden states of the units below are directly connected to the units above, i.e., $x_{t_{above}} = y_{t_{below}}$ and $y_{t_{below}} = h_{t_{below}}$. The output layer (Eq. 7) lies on top of the stack.

6.1.1 Language modeling

Language modeling is a special case of classification where the model is trained to predict the next word or character in the sentence. At each time step t, the output vector gives the probability distribution of x_t over all the words/characters in the vocabulary, conditioned on the previous words/characters in the sequence, that is, $P[x_t|x_{t-1}, ..., x_1]$. At test time, the probability of a full sequence $\{x_1, ..., x_T\}$ is given by the product of all conditional probabilities as:

$$P\big[\{x_1, ..., x_T\}\big] = P[x_1] \times \prod_{t=2}^{T} P[x_t|x_{t-1}, ..., x_1] \tag{8}$$

The language model can also be used to generate text of arbitrary size by repeatedly sampling characters for the desired number of time steps (for character-level granularity) or until the

special end-of-sentence token is selected[11] (for word-level granularity).

For a character-level language model for instance, T can easily exceed 20 or 25. This greatly amplifies the adverse effects of the well-known *vanishing* and *exploding* gradients problem, which prevents long-range dependencies from being learned[12]. 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.

6.2 LSTM unit

In practice, whenever people use RNNs, they use the LSTM or the GRU unit (see next subsection), as these cells are engineered in a way that allows them to escape vanishing/exploding gradients and keep track of information over longer time periods [11].
As shown in Fig. 8, the two things that change in the LSTM unit compared to the basic RNN unit are (1) the presence of a *cell state* (c_t), which serves as an explicit memory, and (2) how hidden states are computed. With vanilla RNNs, the hidden state is computed with a single layer as $h_t = \tanh(Ux_t + Wh_{t-1} + b)$ (see eq. 6). With the LSTM unit however, the hidden state is computed by four interacting layers that give the network the ability to remember or forget specific information about the preceding elements in the sequence.

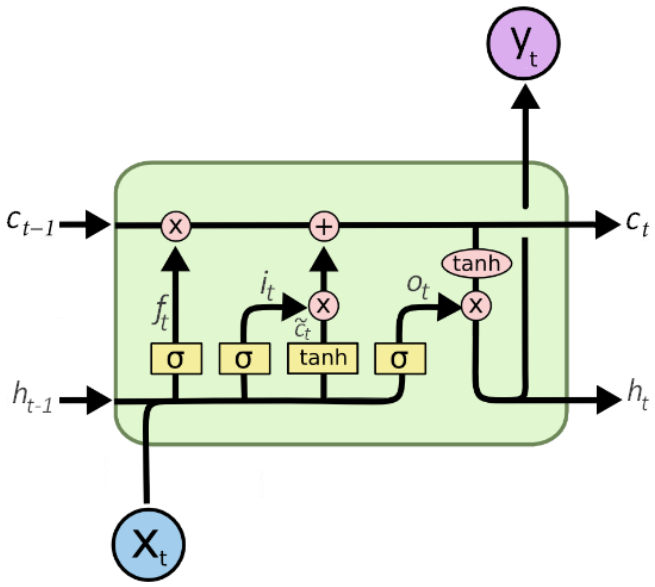

Figure 8: The LSTM unit. Adapted from *Chris Colah's blog*.

6.2.1 Inner layers

The four layers are:

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

[11] see [9] and http://karpathy.github.io/2015/05/21/rnn-effectiveness/
[12] wildml.com/2015/10/recurrent-neural-networks-tutorial-part-3-backpropagation-through-time-and-vanishing-gradients/

Thanks to the elementwise application of the sigmoid function (σ), the forget, input, and output *gate* layers (1, 2, and 4 above) generate vectors whose entries are all comprised between 0 and 1, and either close to 0 or close to 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 forget, input, and output gates are computed in the exact same way, only the parameters vary. The parameters are however shared across all time steps.

6.2.2 Forgetting/learning

By taking into account the new training example x_t and the current hidden state h_{t-1}, the forget gate layer f_t determines how much of the previous cell state c_{t-1} should be forgotten (what fraction of the memory should be freed up), while from the same input, the input gate layer i_t decides how much of the candidate values $\tilde{c}_t$ should be written to the memory, or in other words, how much of the new information should be learned. Combining the output of the two filters *updates* the cell state:

$$c_t = f_t \circ c_{t-1} + i_t \circ \tilde{c}_t \tag{9}$$

Where $\circ$ denotes elementwise multiplication (Haddamard product). This way, important information is not overwritten by the new inputs but is able to be kept alongside them for long periods of time. Finally, the activation h_t is computed from the updated memory, modulated by the output gate layer o_t:

$$h_t = \tanh\big(c_t\big) \circ o_t \tag{10}$$

The output gate allows the unit to only activate when the in-memory information is found to be relevant for the current time step. Finally, as before with the simple RNN, the output vector is computed as a function of the new hidden state:

$$y_t = \text{softmax}(V h_t) \tag{11}$$

6.2.3 Vanilla RNN analogy

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

6.3 Gated Recurrent Unit (GRU)

As shown in Fig. 9, the GRU unit [2] is a simplified LSTM unit with only two gates (reset and update), and where there is no explicit memory c_t.

1. reset gate layer: $r_t = \sigma\big(U_r x_t + W_r h_{t-1} + b_r\big)$
2. update gate layer: $z_t = \sigma\big(U_z x_t + W_z h_{t-1} + b_z\big)$

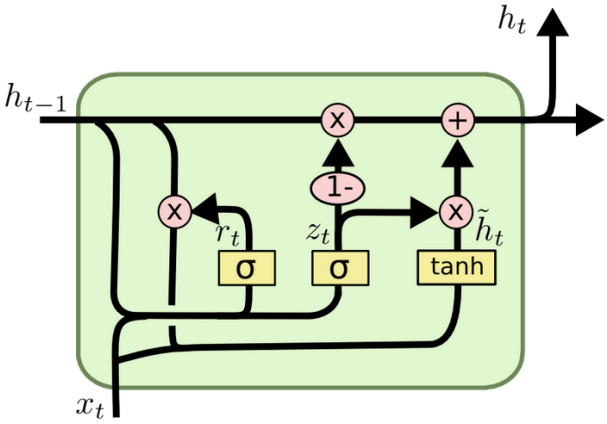

Figure 9: GRU unit. Taken from *Chris Colah's blog*.

The candidate hidden state is computed as:

$$\tilde{h}_t = \tanh\big(U_h x_t + W_h(r_t \circ h_{t-1}) + b_h\big)^{13} \tag{12}$$

When all elements of the reset gate approach zero, information from the previous time steps (stored in h_{t-1}) is discarded, and the candidate hidden state is thus only based on the current input x_t. The new hidden state is finally obtained in a way similar to that of the LSTM cell state, by linearly interpolating between the previous hidden state and the candidate one:

$$h_t = z_t \circ h_{t-1} + (1 - z_t) \circ \tilde{h}_t \tag{13}$$

the only difference is that this time, the update gate z_t serves as the forget gate and determines the fraction of information from the previous hidden state that should be forgotten, and the input gate is *coupled* on the forget gate.

6.4 RNN vs LSTM vs GRU

The basic RNN unit exposes its full hidden state at every time step (see Eq. 6), so as time goes by, the impact of older inputs is quickly replaced by that of the more recent ones. The RNN is therefore not able to remember important features for more than a few steps. Indeed, we have shown previously that a RNN is analogous to a LSTM where for all t, $f_t = \vec{0}$, $i_t = \vec{1}$, and $o_t = \vec{1}$ (we forget everything about the past and learn everything about the present).

On the other hand, thanks to the use of an explicit memory (the cell) and a gating mechanism, the LSTM unit is able to control which fraction of information from the past should be kept in memory (forget gate f_t), which fraction of information from the current input should be written to memory (input gate i_t), and how much of the memory should be exposed to the next time steps and to the units in the higher layers (output gate o_t).

The GRU also features a gating mechanism, but has no explicit memory (no cell state). As a result, the gating mechanism of the GRU is simpler, without output gate: the linear interpolation between the old and the new information is directly injected into the new hidden state without filtering (see Eq. 13). Another difference is that when computing the candidate values, the GRU, via its reset gate r_t, modulates the flow of information coming from the previous activation h_{t-1} (see Eq. 12), while in the LSTM unit, $\tilde{c}_t$ is based on the raw h_{t-1}. Last but not least, in the GRU, the balance between the old and the new information is only made by the update gate z_t

[13] It should be noted that the original formulation of [2] uses $r_t \circ (W_h h_{t-1})$. Here, we adopt the formulation of [4], $W_h(r_t \circ h_{t-1})$. According to [4], the two formulations perform equivalently.

(see Eq. 13), whereas the LSTM unit has two *independent* forget and input gates.

While both the LSTM and GRU units are clearly superior to the basic RNN unit [4], there is no evidence about which one is best [10, 4]. However, since the GRU is simpler, it is easier to implement, more efficient, and has less parameters so it requires less training data.

7 Attention

The attention mechanism [1] was developed in the context of encoder-decoder architectures for Neural Machine Translation (NMT) [2, 24], and rapidly applied to naturally related tasks such as image captioning (translating an image to a sentence) [25], and summarization (translating to a more compact language) [21]. From a high-level, by allowing the decoder to shop for what it needs over multiple vectors, attention relieves the encoder from the burden of having to embed the input into a single fixed-length vector, and thus allows to keep much more information [1].

Today, attention is ubiquitous in deep learning models, and is not used only in encoder-decoder contexts. Notably, attention devices have been proposed for encoders only, to solve tasks such as document classification [27] or representation learning [5]. Such mechanisms are qualified as *self* or *inner* attention.

In what follows, we will start by presenting attention in the original context of encoder-decoder for NMT, using the general framework introduced by [20], and then introduce self-attention.

7.1 Encoder-decoder attention

7.1.1 Encoder-decoder overview

From a very high level, as shown in Fig. 10, the encoder embeds the input into a vector, and the decoder generates some output from this vector.

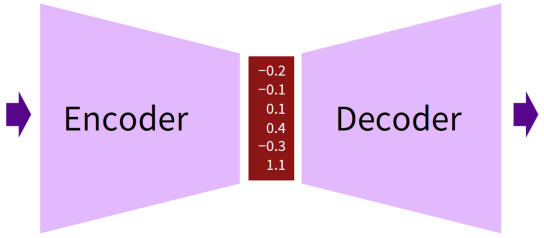

Figure 10: Overview of the encoder-decoder architecture. Taken from https://sites.google.com/site/acl16nmt/home

In Neural Machine Translation (NMT), the input and the output are sequences of words, respectively $x = (x_1, \ldots, x_{T_x})$ and $y = (y_1, \ldots, y_{T_y})$. x and y are usually referred to as the *source* and *target* sentences. When both the input and the output are sequences, encoder-decoder architectures are sometimes called sequence-to-sequence (seq2seq) [24]. Thanks to the fact that encoder-decoder architectures are differentiable everywhere, their parameters θ can be simultaneously optimized with maximum likelihood estimation (MLE) over a parallel corpus. This way of training is called *end-to-end*.

$$\text{argmax}_\theta \left\{ \sum_{(x,y)\in \text{corpus}} \log\, p(y|x;\theta) \right\} \tag{14}$$

Here, the function that we want to maximize is the log probability of a correct translation.

7.1.2 Encoder

The source sentence can be embedded by any model (e.g., CNN, fully connected). Usually for MT though, the encoder is a deep RNN. Bahdanau et al. [1] originally used a *bidirectional* deep RNN. Such a model is made of two deep unidirectional RNNs, with different parameters except the word embedding matrix. The first *forward* RNN processes the source sentence from left to right, while the second *backward* RNN processes it from right to left. The two sentence embeddings are concatenated at each time step t to obtain the inner representation of the bidirectional RNN:

$$h_t = \big[\vec{h_t}; \overleftarrow{h_t}\big] \tag{15}$$

The bidirectional RNN takes into account the entire context when encoding the source words, not just the preceding words. As a result, h_t is biased towards a small window centered on word x_t, while with a unidirectional RNN, h_t is biased towards x_t and the words immediately preceding it. Focusing on a small window around x_t may be advantageous, but does not seem crucial. Indeed, Luong et al. [20] obtained state-of-the-art results with a usual unidirectional deep RNN encoder. In what follows, the hidden states of the encoder will be written $\bar{h}_t$. They are sometimes called *annotations* in the literature.

7.1.3 Decoder

While different models can be used as the encoder, in NMT the decoder is usually a unidirectional RNN because this model is naturally adapted to the sequential nature of the generation task, and is usually deep (stacking). The decoder generates each word of the target sentence one step at a time.

Key idea. Making the decoder use only the last annotation h_{T_x} produced by the encoder to generate output forces the encoder to fit as much information as possible into h_{T_x}. Since h_{T_x} is a single fixed-size vector, its capacity is limited, so some information is lost. On the other hand, the attention mechanism allows the decoder to consider the entire sequence $\big(h_1, \dots, h_{T_x}\big)$ of annotations produced by the encoder at each step of the generation process. As a result, the encoder is able to keep much more information by distributing it among all its annotations, knowing that the decoder will be able to decide later on which vectors it should pay attention to.

More precisely, the target sentence $y = (y_1, \dots, y_{T_y})$ is generated one word y_t at a time based on the distribution:

$$P\big[y_t|\{y_1, ..., y_{t-1}\}, c_t\big] = \text{softmax}\big(W_s\tilde{h}_t\big) \tag{16}$$

where $\tilde{h}_t$, the *attentional* hidden state, is computed as:

$$\tilde{h}_t = \tanh\big(W_c\big[c_t; h_t\big]\big) \tag{17}$$

h_t is the hidden state of the decoder (hidden state of the top layer, when the decoder is a stacking RNN) and provides information about the previously generated target words $\{y_1, ..., y_{t-1}\}$, c_t is the source context vector, and $\big[;\big]$ is concatenation. W_s and W_c are matrices of trainable parameters. Biases are not shown for simplicity. As shown in Fig. 11, the context vector c_t can be computed in two ways: *globally* and *locally*. We describe each approach in the next two subsections.

A note on beam search. Trying all possible combinations of words in the vocabulary to find the target sentence with highest joint probability is intractable. But on the other hand, generating y in a purely greedy way, i.e., by selecting the most likely word every time, is highly suboptimal. In practice, a certain number K of candidate translations are explored with *beam*

search, a heuristic search algorithm [7]. Large values of K generate better target sentences, but decrease decoding speed.

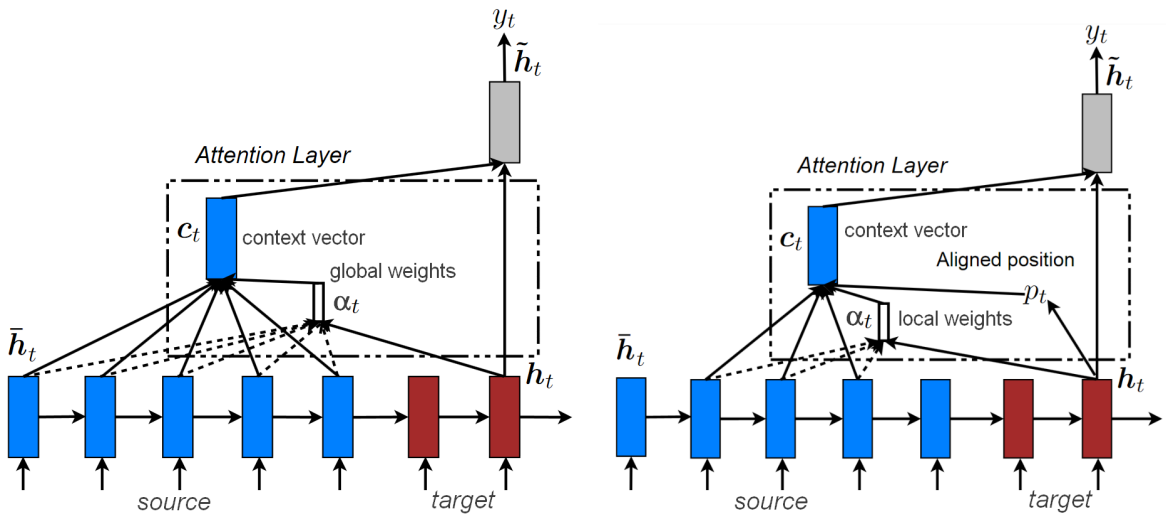

Figure 11: Global (left) vs local attention (right). Adapted from [20].

7.1.4 Global attention

Here, the context vector c_t is computed as a weighted sum of the *full list* of annotations $\bar{h}_i$ of the source sentence (i.e., the hidden states of the encoder). There are T_x annotations. Each one is a vector of size the number of units in the hidden layer of the encoder. c_t has same size as any annotation. The size of the alignment vector α_t is equal to the size T_x of the source sentence, so it is *variable*.

$$c_t = \sum_{i=1}^{T_x} \alpha_{t,i} \bar{h}_i \tag{18}$$

The alignment vector α_t is computed by applying a softmax to the output of an *alignment* operation (`score()`) between the current target hidden state h_t and all source hidden states $\bar{h}_i$'s:

$$\alpha_{t,i} = \frac{\exp\big(\text{score}(h_t, \bar{h}_i)\big)}{\sum_{i'=1}^{T_x} \exp\big(\text{score}(h_t, \bar{h}_{i'})\big)} \tag{19}$$

In other words, α_t is a probability distribution over all source hidden states (its coefficients are all between 0 and 1 and sum to 1), and indicates which words in the source sentence are the most likely to help in predicting the next word. `score()` can in theory be any comparison function. Luong et al. [20] experimented with the dot product ($\text{score}(h_t, \bar{h}_i) = h_t^\top \bar{h}_i$), a more general formulation with a matrix of parameters ($\text{score}(h_t, \bar{h}_i) = h_t^\top W_\alpha \bar{h}_i$), and a fully connected layer. They found that *dot* works better for global attention while *general* is superior for local attention. A summary of global attention is provided in Fig. 12.

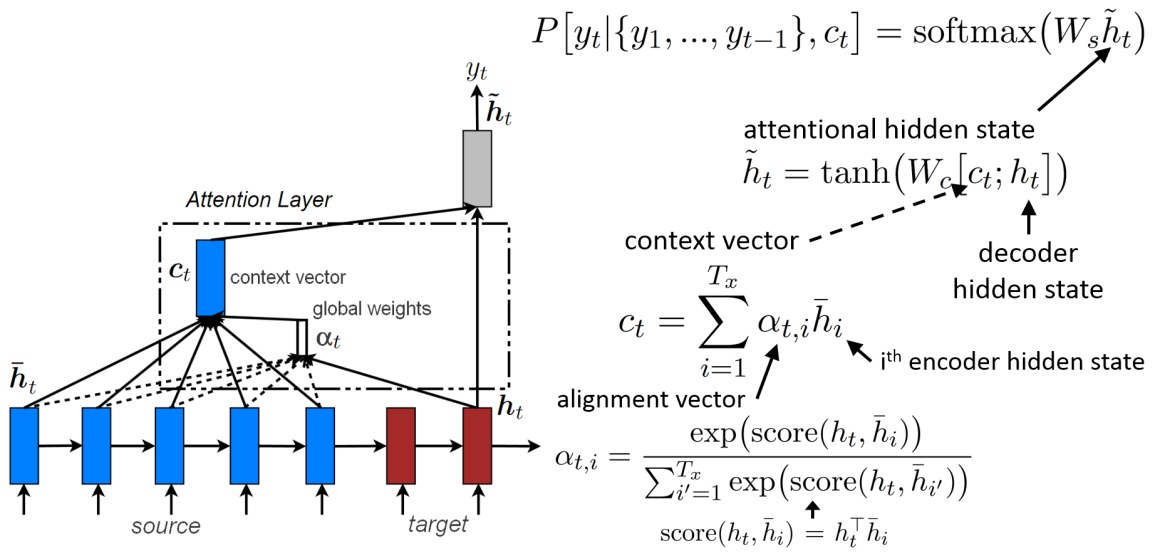

Figure 12: Summary of the *global attention* mechanism [20].

7.1.5 Local attention

Considering all words in the source sentence to generate every single target word is expensive, and may not be necessary. To remediate this issue, Luong et al. [20] proposed to focus only on a small window of annotations of fixed size $2D+1$:

$$c_t = \sum_{i=p_t-D}^{p_t+D} \alpha_{t,i}\bar{h}_i \tag{20}$$

D is prescribed by the user, and the position p_t where to center the window is either set to t (*monotonic* alignment) or determined by a differentiable mechanism (*predictive* alignment) based on information about the previously generated target words $\{y_1,...,y_{t-1}\}$ stored in h_t:

$$p_t = T_x \cdot \sigma\big(v_p^\top \tanh(W_p h_t)\big) \tag{21}$$

Where T_x is the length of the source sentence, σ is the sigmoid function, and v_p and W_p are trainable parameters. Alignment weights are computed like in the case of global attention (Eq. 19), with the addition of a Normal distribution term centered on p_t and with standard deviation $D/2$:

$$\alpha_{t,i} = \frac{\exp\big(\text{score}(h_t,\bar{h}_i)\big)}{\sum_{i'=p_t-D}^{p_t+D}\exp\big(\text{score}(h_t,\bar{h}_{i'})\big)} \exp\Big(-\frac{(i-p_t)^2}{2(D/2)^2}\Big) \tag{22}$$

Note that $p_t \in \mathbb{R} \cap \big[0, T_x\big]$ and $i \in \mathbb{N} \cap \big[p_t - D, p_t + D\big]$. The addition of the Gaussian term makes the alignment weights decay as i moves away from the center of the window p_t, i.e., it gives more importance to the annotations near p_t. Also, unlike with global attention, the size of α_t is fixed and equal to $2D+1$, as only the annotations within the window are taken into account. Local attention can actually be viewed as global attention where alignment weights are multiplied by a truncated Normal distribution (i.e., that returns zero outside the window). A summary of local attention is provided in Fig. 13.

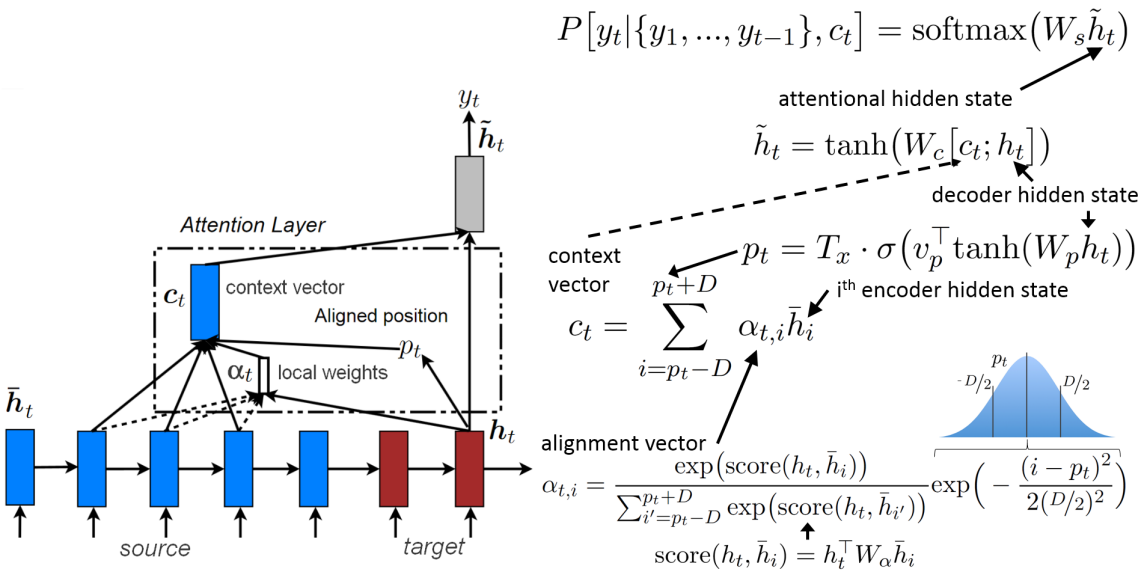

Figure 13: Summary of the *local attention with predictive alignment* mechanism [20].

7.2 Self-attention

We are here in a simpler setting with a single RNN encoder taking as input a sequence $\big(x_1,\ldots,x_T\big)$ of length T. As usual, the RNN maps the input sequence to a sequence of annotations $\big(h_1,\ldots,h_T\big)$. The goal is exactly the same as with attention in the encoder-decoder context: rather than considering the last annotation h_T as a comprehensive summary of the entire sequence, which is prone to information loss, a new hidden representation is computed by taking into account the annotations at *all* time steps. To this purpose, the self-attention or inner attention mechanism emerged in the literature in 2016/2017, with, e.g., [27, 18]. In what follows we use the formulation of [27].

As shown in Eq. 23, annotation h_t is first passed to a dense layer. An alignment coefficient α_t is then derived by comparing the output u_t of the dense layer with a trainable context vector u (initialized randomly) and normalizing with a softmax. The attentional vector s is finally obtained as a weighted sum of the annotations.

$$\begin{aligned} u_t &= \tanh(Wh_t) \\ \alpha_t &= \frac{\exp(\text{score}(u_t,u))}{\sum_{t'=1}^{T}\exp(\text{score}(u_{t'},u))} \\ s &= \sum_{t=1}^{T}\alpha_t h_t \end{aligned} \tag{23}$$

score can in theory be any alignment function. A straightforward approach is to use $\text{score}(u_t,u) = u_t^\top u$. The context vector can be interpreted as a representation of the optimal word, on average. When faced with a new example, the model uses this knowledge to decide which word it should pay attention to. During training, through backpropagation, the model updates the context vector, i.e., it adjusts its internal representation of what the optimal word is.

7.2.1 Difference with seq2seq attention

The context vector in the definition of self-attention above has nothing to do with the context vector used in seq2seq attention! In seq2seq, the context vector c_t is equal to the weighted sum $\sum_{i=1}^{T_x}\alpha_{t,i}\bar{h}_i$, and is used to compute the attentional hidden state as $\tilde{h}_t = \tanh\big(W_c\big[c_t;h_t\big]\big)$. In

self-attention however, the context vector is simply used as a replacement for the hidden state of the decoder when performing the alignment with score(), since there is no decoder. So, in self-attention, the alignment vector α indicates the *similarity of each input word with respect to the optimal word (on average)*, while in seq2seq attention, α indicates the *relevance of each source word in generating the next element of the target sentence.*

7.2.2 Hierarchical attention

A simple, good example of how self-attention can be useful in practice is provided by the architecture illustrated in Fig. 14. In this architecture, the self-attention mechanism comes into play twice: at the word level, and at the sentence level. This approach makes sense for two reasons: first, it matches the natural hierarchical structure of documents (words → sentences → document). Second, in computing the encoding of the document, it allows the model to first determine which words are important in each sentence, and then, which sentences are important overall. Through being able to re-weigh the word attentional coefficients by the sentence attentional coefficients, the model captures the fact that a given instance of word may be very important when found in a given sentence, but another instance of the same word may not be that important when found in another sentence.

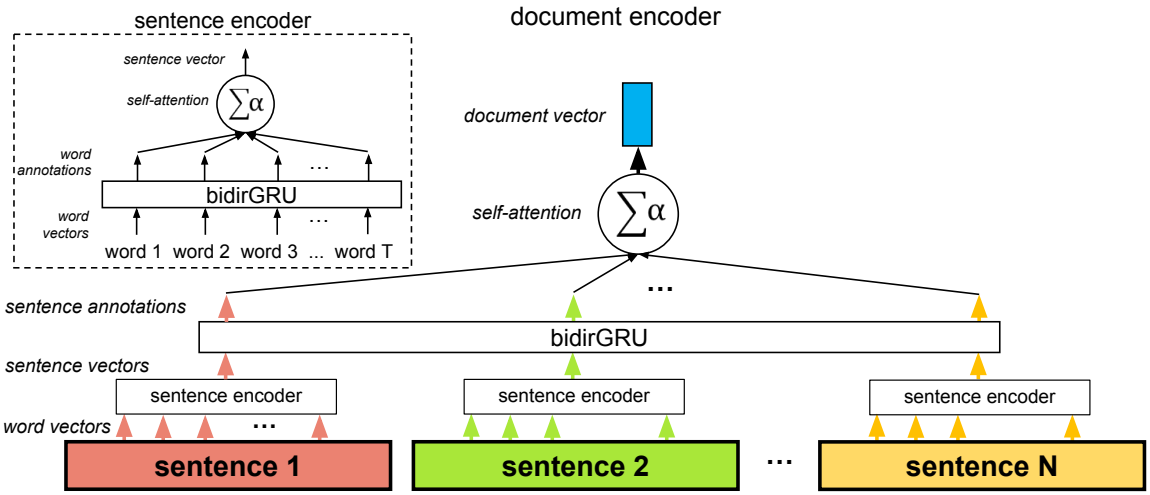

Figure 14: Hierarchical attention architecture [27].

References

[1] Bahdanau, Dzmitry, Kyunghyun Cho, and Yoshua Bengio. "Neural machine translation by jointly learning to align and translate." arXiv preprint arXiv:1409.0473 (2014). 2, 14, 15

[2] 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, 12, 13, 14

[3] 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. 6

[4] Chung, J., Gulcehre, C., Cho, K., & Bengio, Y. (2014). Empirical evaluation of gated recurrent neural networks on sequence modeling. arXiv preprint arXiv:1412.3555. 2, 13, 14

[5] Conneau, A., Kiela, D., Schwenk, H., Barrault, L., & Bordes, A. (2017). Supervised learning of universal sentence representations from natural language inference data. arXiv preprint arXiv:1705.02364. 14

[6] Elman, J. L. (1990). Finding structure in time. Cognitive Science, 14:2, 179-211. 9

[7] Freitag, Markus, and Yaser Al-Onaizan. "Beam search strategies for neural machine translation." arXiv preprint arXiv:1702.01806 (2017). 16

[8] Goldberg, Y. (2015). A primer on neural network models for natural language processing. Journal of Artificial Intelligence Research, 57, 345-420. 2, 3, 9

[9] Graves, A. (2013). Generating sequences with recurrent neural networks. arXiv preprint arXiv:1308.0850. 11

[10] Greff, Klaus, et al. "LSTM: A search space odyssey." IEEE transactions on neural networks and learning systems 28.10 (2017): 2222-2232. 14

[11] Hochreiter, S., Schmidhuber, J. (1997). Long short-term memory. Neural computation, 9(8), 1735 11

[12] 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. 4

[13] Johnson, R., Zhang, T. (2015). Effective Use of Word Order for Text Categorization with Convolutional Neural Networks. To Appear: NAACL-2015, (2011). 2, 8

[14] 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. 2, 3, 4

[15] Krizhevsky, Alex, Ilya Sutskever, and Geoffrey E. Hinton. "Imagenet classification with deep convolutional neural networks." Advances in neural information processing systems. 2012. 4

[16] 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. 2, 4

[17] Li, J., Chen, X., Hovy, E., and Jurafsky, D. (2015). Visualizing and understanding neural models in nlp. arXiv preprint arXiv:1506.01066. 8

[18] Lin, Zhouhan, et al. "A structured self-attentive sentence embedding." arXiv preprint arXiv:1703.03130 (2017). 18

[19] Lipton, Zachary C., John Berkowitz, and Charles Elkan. "A critical review of recurrent neural networks for sequence learning." arXiv preprint arXiv:1506.00019 (2015). 9

[20] Luong, Minh-Thang, Hieu Pham, and Christopher D. Manning. "Effective approaches to attention-based neural machine translation." arXiv preprint arXiv:1508.04025 (2015). 2, 14, 15, 16, 17, 18

[21] Rush, Alexander M., Sumit Chopra, and Jason Weston. "A neural attention model for abstractive sentence summarization." arXiv preprint arXiv:1509.00685 (2015). 14

[22] Simonyan, K., Vedaldi, A., and Zisserman, A. (2013). Simonyan, Karen, Andrea Vedaldi, and Andrew Zisserman. "Deep inside convolutional networks: Visualising image classification models and saliency maps." arXiv preprint arXiv:1312.6034 (2013). arXiv preprint arXiv:1312.6034. 8

[23] Springenberg, Jost Tobias, et al. "Striving for simplicity: The all convolutional net." arXiv preprint arXiv:1412.6806 (2014). 4

[24] Sutskever, Ilya, Oriol Vinyals, and Quoc V. Le. "Sequence to sequence learning with neural networks." Advances in neural information processing systems. 2014. 2, 6, 14

[25] Xu, K., Ba, J., Kiros, R., Cho, K., Courville, A., Salakhudinov, R., ... & Bengio, Y. (2015, June). Show, attend and tell: Neural image caption generation with visual attention. In International Conference on Machine Learning (pp. 2048-2057). 14

[26] Zhang, Ye, and Byron Wallace. "A sensitivity analysis of (and practitioners' guide to) convolutional neural networks for sentence classification." arXiv preprint arXiv:1510.03820 (2015). 2, 3, 5, 6

[27] Yang, Zichao, et al. "Hierarchical attention networks for document classification." Proceedings of the 2016 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies. 2016. 14, 18, 19