50.021 -AI

Alex

Week 06: Recurrent neural nets

[The following notes are compiled from various sources such as textbooks, lecture materials, Web resources and are shared for academic purposes only, intended for use by students registered for a specific course. In the interest of brevity, every source is not cited. The compiler of these notes gratefully acknowledges all such sources.]

Recurrent neural nets

Recurrent neural nets is a tool for processing sequence data based on neural networks.

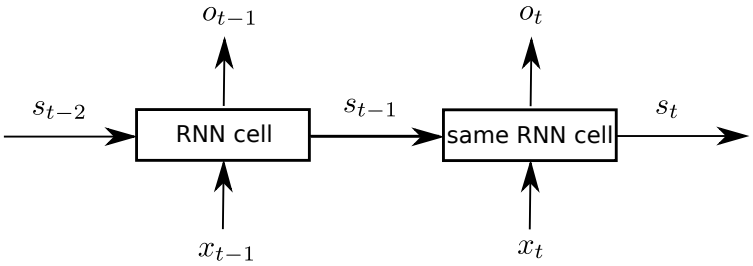
- time series $x = (x_1, x_2, \ldots, x_T)$ like stock prices.
- sentences $x = (x_1 = \text{This}, x_2 = \text{is}, x_3 = \text{a}, x_4 = \text{hungry-looking}, x_5 = \text{fish}, x_6 = !)$ or counts of passengers in a bus – which may change at every station (or the age distribution of the passengers).

The coarse idea is: We use the t in x_t as sequence index notation. t does not need to be a time. [1]

[1] What is t in the example of the text or the bus counts

We want to predict something over a sequence. The prediction can be a number or another sequence. One example is to predict whether to buy, hold or sell at the next time step (prediction would be here 3 numbers - for probabilities), another example is: translate one sentence into another language. Then the prediction would be a sequence – the words of the correct translation

The first question is: How to use recurrent neural nets to process sequence data? Lets look at one Recurrent neural net cell.



- Recurrent neural net cells have an internal state s_t, often encoded by some vector.
- At step t the element x_t of the sequence x is fed in, and the internal state is updated: we compute the new interal state s_t from the old one s_{t-1}, and possibly an output is produced: o_t.

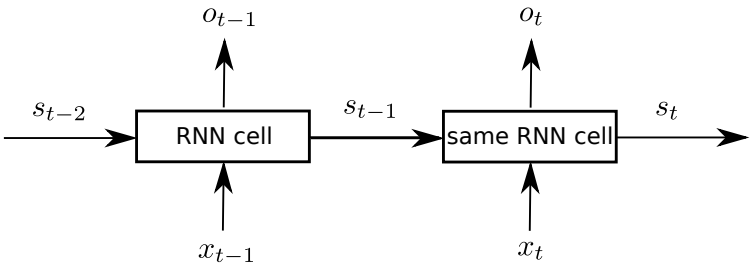
$$s_t = f(s_{t-1}, x_t)$$
$$o_t = g(x_t, s_t)$$
or
$$o_t = g(x_t, s_{t-1})$$

f, g are functions specific to the implementation of recurrent neural nets.

A very simple example is: suppose x_t is a vector in 10 dimensions, the state s_t is a vector in 30 dimensions, the output o_t is a vector in 5 dimensions. Then one could define as equations

$$s_t = tanh(W_{ss}s_{t-1} + W_{xs}x_t)$$
$$W_{ss}.shape = (30, 30), W_{xs}.shape = (30, 10)$$
$$o_t = tanh(W_{xo}x_t + W_{so}s_{t-1})$$
$$W_{xo}.shape = (5, 10), W_{so}.shape = (5, 30)$$

The *tanh* is applied element-wise. The idea here is same as for other neural networks: We apply weights W that can be learned, then apply an activation function on top. The model is a linear function plus non-linear activation. We learn the weights W during training.



The difference to feedforward networks before: You have learnable weight parameters as with convolutional networks, but now also a persistent state s_t – which depends on the history of inputs and on your learnable parameters.

The state is carried forward from one element in the sequence to another.

Famous examples are LSTM (Long Term Short Term memory, Hochreiter and Schmidthuber, 1997), and GRU (gated recurrent unit, Kyunghyun Cho, 2014).

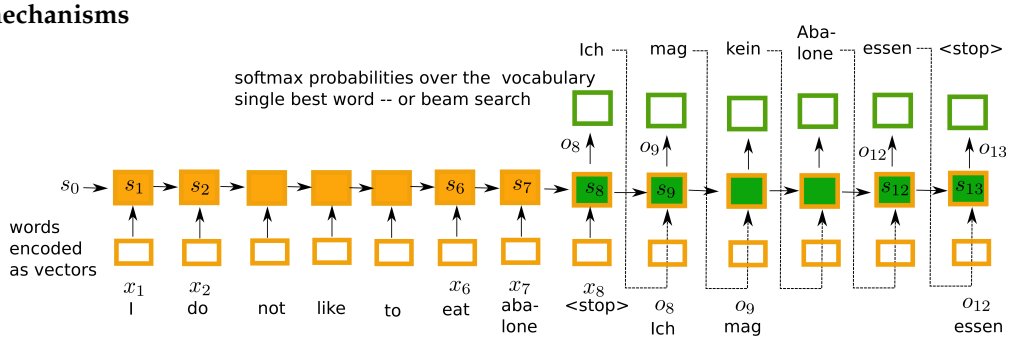
sequence to sequence learning

The Problem of sequence to sequence learning: sequence to sequence learning is to predict a sequence as output given a sequence as input.

One example for it is **neural machine translation**. Input is a sentence in one language. Output is the translation in another language.

The challenge is that the input and output sequences may have varying length, and different length from each other. The best translation often has not the same number of words as the original sentence.

A simple neural machine translation model without attention mechanisms



prediction time: How to use such a structure to make a prediction?

- We add a stop word as symbol for end of sequence.
- **encoding phase**: We feed in the sentence word by word. We ignore outputs during this phase.
- **decoding phase**: We feed in the stop word, from now on we expect outputs. We fetch the first output, and use it as the next input.
  How to be able to generate a second output? In order to obtain a new output, we must update the internal state s_t by feeding in some input. The core idea: use the last output as new input.
- in the decoding phase we use the output of the previous step as input for the next step. We terminate when output is the stop sequence. [2]

  What does the model output? The softmax encodes

$$P(o_t|o_s, o_{s+1}, \ldots, o_{t-1}, \text{Input sentence})$$

It is the probability to see word o_t at position t given the input sentence $(x_1, x_2, \ldots)$ and given that we have fetched before the words $o_s, o_{s+1}, \ldots, o_{t-1}$.

In the simplest way the fetched output is the word with the highest probability

$$\hat{o}_t = \text{argmax}_{\text{words } o \text{ in vocabulary}} P(o|o_s, o_{s+1}, \ldots, o_{t-1}, \text{Input sentence})$$

But please see the section on beam search

Training time:

How can that be used to train? Suppose we have in our training dataset for every input sentence one correct output sentence. A sentence is a sequence of words. We use as outputs in the simplest case softmax probabilities for every possible word.

[2] I have hidden here what means to fetch a word. One needs to say what the meaning of a softmax is in the above graphic. The softmax at time step t is the probability to observe a word o given that we have fed in the input sequence $x_1, \ldots, x_9$ and given that we *have selected/fetched already* output words $o_1, \ldots, o_{t-1}$ until the time step $t-1$. In the simplest method *fetching a word at time step t* means to select as o_t the word that has highest output probability in the softmax for o_t – but you can do better. See section Other Tricks below.

Then we can measure for every output word the probability $P(o_t|o_s, o_{s+1}, \ldots, o_{t-1}, \text{Input sentence})$, and use a loss similar to the cross entropy loss.

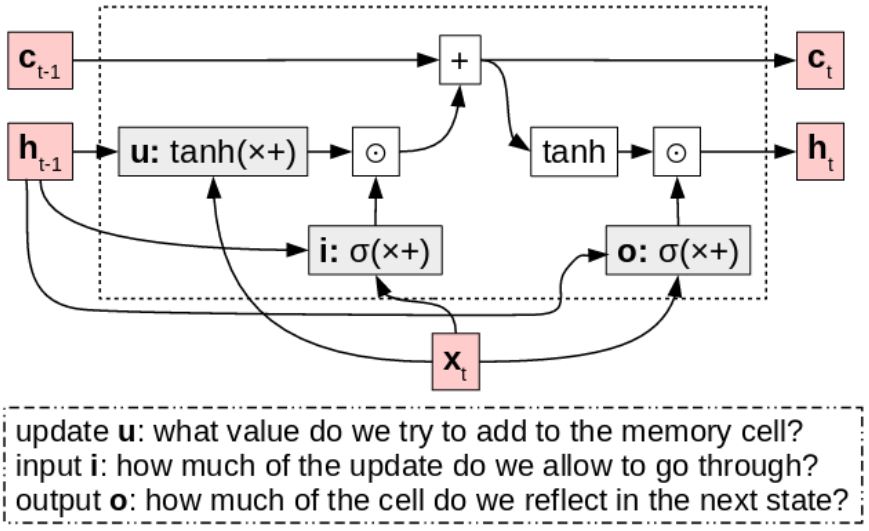
Cross entropy for object classes had low loss if the probability for the output with the ground truth class label was high. Now replace object classes by words, and ground truth class label by ground truth word. Same thing.

vanishing/exploding gradients and LSTM

Training recurrent neural nets can suffer from the problem of gradients over time exploding or vanishing when using backpropagation. Problem is: we update the internal state s_t over many time steps. If $ds_t/ds_{t-1} \neq 1$, then the gradients may either grow unbounded step by step or shrink to zero.

LSTM was designed to be more robust against it. It has a hidden state s and a memory cell c. For the memory cell $dc_t/dc_{t-1} = 1$.

Image credit: Graham Neubigs Tutorial (see below)



$$
\begin{aligned}
\boldsymbol{u}_t &= \tanh(W_{xu}\boldsymbol{x}_t + W_{hu}h_{t-1} + \boldsymbol{b}_u) \\
\boldsymbol{i}_t &= \sigma(W_{xi}\boldsymbol{x}_t + W_{hi}h_{t-1} + \boldsymbol{b}_i) \\
\boldsymbol{o}_t &= \sigma(W_{xo}\boldsymbol{x}_t + W_{ho}h_{t-1} + \boldsymbol{b}_o) \\
\boldsymbol{c}_t &= \boldsymbol{i}_t \odot \boldsymbol{u}_t + \boldsymbol{c}_{t-1} \\
\boldsymbol{h}_t &= \boldsymbol{o}_t \odot \tanh(\boldsymbol{c}_t).
\end{aligned}
$$

The seq element at time t x_t is fed in.

- The seq element is used to compute an update $u_t = Fun_1(x_t, h_{t-1})$ which depends on the last hidden state and the seq element. The update vector is in every component bounded by $[-1, +1]$ bcs it is the output of a tanh.
- The seq element is used to compute an input vector. $i_t = Fun_1(x_t, h_{t-1})$. The input vector is in every component is a bounded by $[0, 1]$ bcs it is the output of a sigmoid.
- input and update are element-wise multiplied and used to change the content of the memory cell c. See: by that element-wise multiplication the input acts as a weight. $c_t = i_t \odot u_t + c_{t-1}$.
- the output vector o_t is a function of the seq element x_t and the old hidden state h_{t-1}.
- the output vector is used to compute a new hidden state h_t, using the updated memory cell $h_t = o_t \odot tanh(c_t)$
- one can add a forget gate to allow the memory cell to clear its contents partially. the equation $c_t = i_t \odot u_t + c_{t-1}$. changes to $c_t = i_t \odot u_t + f_t \odot c_{t-1}$. where f_t is a vector of forget weights.
- all the activation functions are logistic functions (map to $[0, 1]$) or tanh (map to $[-1, +1]$)
- memory may exist on different time-scales (like when reading a book. Information can be relevant from previous chapters and from the previous paragraph.)
- alternatives: gated recurrent unit (GRU), bidirectional recurrent neural networks, models with attention modeling.

Disclaimer: The goal of this part is to show you some directions that are important in neural machine translation - so that you know where to look for when you really need it. The goal is not to make a deeply involved topic, that requires months and tons of papers for being understood, understandable in one lecture!

word embeddings

We need to feed in words. Words are often encoded by word embeddings, that are mapping of words to vectors.

- textual words are usually encoded as vectors. Word embeddings like *Word2Vec* `https://code.google.com/archive/p/word2vec/`, *GloVe* `https://nlp.stanford.edu/projects/glove/`, *Conceptnet Numberbatch* `https://github.com/commonsense/`

`conceptnet-numberbatch` map words into a vector space such that similar words are close by euclidean distance.

Naturally you can list for every embedded word the most similar vectors. The embedding of *ladybug* should have words close by like: *bee,bug, butterfly, insect*, and words like *ocean,AI,grammar,toothbrush* should be more far away from that.

One natural way of measuring similarity in a vector space is the inner product between two vectors!

Examples for similarities can be also evaluated for example by pairwise analogies:

Queen to Woman =? to Man.

Vector spaces are not suitable for dividing vectors, but you can subtract vectors and compute similarities between differences of vectors

$$enc(Princess) - enc(Woman)$$
$$enc(Frog) - enc(Man)$$

You can evaluate similarities between differences of such vectors for measuring qualities of embeddings.

- To see similarities of word embeddings, you can look at for example: `https://blog.conceptnet.io/2016/05/25/conceptnet-numberbatch-a-new-name-for-the-best-word-embedd` - they compute a simple similarity: the inner product between the name of an actor ("Cumberbatch") and the word "actor" and then a similarity between the
- an example how to train such word embeddings: `https://nlp.stanford.edu/pubs/glove.pdf`
- Be careful when using text corpora for training of e.g. chatbots – many datasets have biases: `https://blog.conceptnet.io/2017/04/24/conceptnet-numberbatch-17-04-better-less-stereotyped-word-vectors/`. An algorithm that you train with those, will easily pick them up `https://en.wikipedia.org/wiki/Tay_(bot)`.

  That can a risk for your startup – do you want to be in the news for the company with a chatbot insulting users? It will not kill Microsoft for sure. Think about such things before blindly deploying AI hype outcomes.

- A side question: do you really want chatbots for talking to elderly people? Do you really want chatbots for customer service? Do you want fully automated diagnosis for suspected cancer cases?

Fixed versus open vocabulary

Neural machine translation relies on a fixed dictionary. Texts do not follow such rule. Grammar can create many similar words from one word: the verb *love* can appear as *loved* or *loving*. Its way worse in french, german, russian, as there pronouns and nouns can change endings, too (and verb endings are much more rich, note also all the irregular verbs).

There are approaches to deal with the problem that one can have many variations of the same verb/noun/etc:

Examples are: word stemming, falling back to a dictionary to look up endings, applying grammar rules, learning subword units `https://arxiv.org/pdf/1508.07909.pdf`. There are many tricks involved.[3]

[3] Profs Lu Wei and Zhang Yue know much more than me on such topics!

Other tricks

So far I have omitted what it means to fetch an outputs. It helps to understand what the softmax output represents in the above graphic. The softmax encodes

$$P(o_t|o_1,\ldots,o_{t-1},\text{Input sentence})$$

It is the probability to see word o_t at position t given the input sentence and given that we have fetched before the words $o_1,\ldots,o_{t-1}$

We are not looking for a single word with highest probability. Actual goal is: We want the sentence $O = (o_1,\ldots,O_R = \langle stop\rangle)$ with highest probability (given the input sentence that was to be translated). The probability of a whole sentence O is given as

$$P(O|\text{Input sentence}) = \prod_{t=1}^{S} P(o_t|o_s,o_{s+1},\ldots,o_{t-1},\text{Input sentence})$$

So our goal is:

$$\hat{O} = \text{argmax}_{(o_1,\ldots,o_R)} \prod_{t=1}^{S} P(o_t|o_s,o_{s+1},\ldots,o_{t-1},\text{Input sentence})$$

For finding it one has several heuristics. The simplest one is **greedy search:**

given $o_s,o_{s+1},\ldots,o_{t-1}$, we select o_t by choosing the word that maximizes $P(o_t|o_s,o_{s+1},\ldots,o_{t-1}$

A better variant is so called **beam search** with beam length b. We keep at every step not the single best sequence, but instead b many sequences $o_1,\ldots,o_{t-1}$ in a set – namely those sequences that have highest probability of observing them. For every sequence in the set

we select at step t some of their own best o_t as above – in order to create a new candidate sequence $o_1, \ldots, o_{t-1}, o_t$ for the next iteration [4]. This can work better compared to greedy search (it can be that greedy search decides not to choose a word that has low probability, but all subsequent probabilities are high for a sequence that contains that omitted word).

[4] it can happen that one sequence can create more than one candidate for the set at the next iteration

Consider the following example: we want to find sequences of length 2 over an alphabet $\{a, b, c\}$. We know: $P(O) = P(o_1)P(o_2|o_1)$.

$$p(o_1 = a) = 0.1, p(o_1 = b) = 0.3, p(o_1 = c) = 0.6$$
$$p(o_2 = a|o_1 = c) = 0.3, p(o_1 = b|o_2 = c) = 0.4, p(o_2 = c|o_1 = c) = 0.3$$
$$p(o_2 = a|o_1 = b) = 1, p(o_1 = b|o_2 = b) = 0, p(o_2 = c|o_1 = b) = 0$$

Here greedy search selects $o_1 = c$ and $o_2 = b$, resulting probability is $P(o_1)P(o_2|o_1) = 0.6 \cdot 0.4 = 0.24$

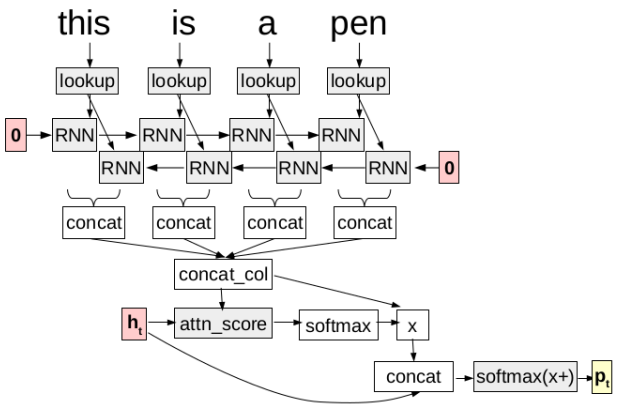
Beam search with beam size $b \geq 2$ will find $o_1 = b$ and $o_2 = a$, resulting probability is $P(o_1)P(o_2|o_1) = 0.3 \cdot 1 = 0.3$. This is higher. It will find it because it will keep the second best hypothesis $o_1 = b$ at iteration 1.

Neural attention models

This is out of class. One of the most read papers is:

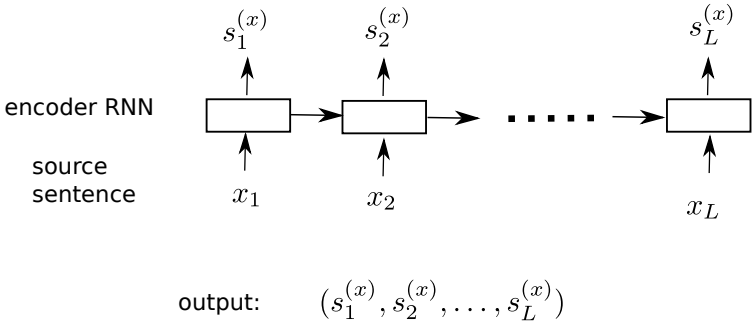
Dzmitry Bahdanau, Kyunghyun Cho, Yoshua Bengio, *Neural Machine Translation by Jointly Learning to Align and Translate*, arxiv.org
`https://arxiv.org/abs/1409.0473`

Typical graph-based explanations can be somewhat confusing:



Lets go for a bit different explanaiton

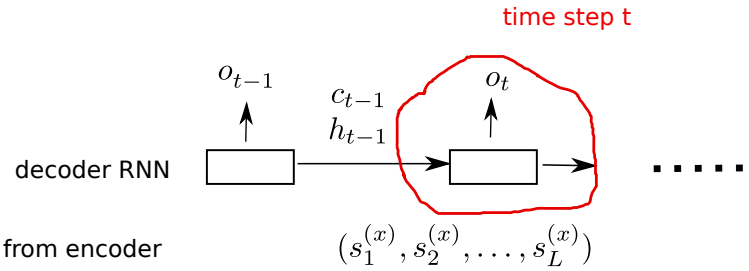
- Suppose you want to translate a source sentence $x_1, \ldots, x_L$. Target sentence (the translation) should be $o_L, \ldots o_{L+M}$ – it has a different length.
- You do a separate encoding part – which reads in and processes the source sentence $x_1, \ldots, x_L$, and a separate decoding part, which produces the target sentence $o_L, \ldots o_{L+M}$.
- **in the encoding part:** we produce for every element x_t in the sequence one vector $s_t^{(x)}$ [5]. The set of all these vectors $(s_1^{(x)}, s_2^{(x)}, \ldots, s_L^{(x)})$ has a variable length L - equal to the length of the source sentence.



The output of the encoding part is the set of all encoder states $(s_1^{(x)}, s_2^{(x)}, \ldots, s_L^{(x)})$. **encoding done.**

[5] in above paper a bi-directional neural net is used to produce such vector $s_t^{(x)}$ at every time step, but this fact is not important for understanding the idea of attention models. Most attention model explanations are mixed up with the bidirectional RNN.

- **in the DEcoding part:**



from last decoder time step: $o_{t-1}, c_{t-1}, h_{t-1}$

1. decoder state update $h_t = f(c_{t-1}, o_{t-1}, h_{t-1})$

2. compute similarities to encoder states

$$a^{(t)} = (a_1^{(t)}, a_2^{(t)}, \ldots, a_L^{(t)}) = (sim(h_t, s_1^{(x)}), sim(h_t, s_2^{(x)}), \ldots, sim(h_t, s_L^{(x)}))$$

3. compute weights for encoder states (based on similarities)

$$\alpha^{(t)} = (\alpha_1^{(t)}, \alpha_2^{(t)}, \ldots, \alpha_L^{(t)}) = softmax(a_1^{(t)}, \ldots a_L^{(t)})$$

4. compute new weighted state from encoder

$$c_t = \alpha_1^{(t)} s_1^{(x)} + \alpha_2^{(t)} s_2^{(x)} + \ldots + \alpha_L^{(t)} s_L^{(x)}$$

5. compute new output probabilities for output

$$p(o_t) = softmax(W[h_t, c_t] + b)$$

- Goal: **input a vector of fixed dimensionality** into the decoder (the decoder produces the target sentence words o_t) at time step t

Note: each of the $s_t^{(x)}$ has the same dimensionality, but their number L is variable.

Suppose we can obtain weights $\alpha_1^{(t)}, \ldots, \alpha_L^{(t)}$ – which are in $[0, 1]$ and sum up to one, then

$$c_t = \alpha_1^{(t)} s_1^{(x)} + \alpha_2^{(t)} s_2^{(x)} + \ldots + \alpha_L^{(t)} s_L^{(x)}$$
$$c_0 = 0 \text{ Initialization}$$

would give us a fixed dimensionality vector $\boldsymbol{c_t}$
– which is what we want.

$\boldsymbol{c_t}$ would be a weighted sum of encoder states. The weights $\alpha_1^{(t)}, \ldots, \alpha_L^{(t)}$ provides an attention to each of the encoder states at decoding time t. That is the attention. How to achieve that?

The rough idea is: to compute such weights $\alpha_i^{(t)}$ from a similarity

measure between each of the encoder states $s_i^{(x)}$, and the current state of the decoder h_t^{dec}.

The more similar the current state of the decoder is to some encoder state $s_i^{(x)}$, the more weight $\alpha_i^{(t)}$ should be put on that encoder state $s_i^{(x)}$ in the resulting sum

$$c_t = \alpha_1^{(t)} s_1^{(x)} + \alpha_2^{(t)} s_2^{(x)} + \ldots + \alpha_L^{(t)} s_L^{(x)}$$

- We have at the start of time step t in the decoder we have available:
  - the internal state of the decoder from the last time step $h_{t-1}^{(dec)}$
  - the word embedding of the last output word o_{t-1}
  - the fixed dimensionality vector from the last step c_{t-1}.
  - the set of all encoder states $(s_1^{(x)}, s_2^{(x)}, \ldots, s_L^{(x)})$
- at first we **update the internal state of the decoder**:

$$h_t^{(dec)} = f(h_{t-1}^{(dec)}, enc(o_{t-1}), c_{t-1})$$

This uses: the internal decoder state $h_{t-1}^{(dec)}$ from the last time step, the word embedding of the last output word o_{t-1} [6], some function f.

[6] or the softmax vector over outputs from the last step

- now we **compute a similarity between** the **updated state** $h_t^{(dec)}$ and each of **the encoder states**

$$\begin{aligned} a_1^{(t)} &= sim(h_t^{(dec)}, s_1^{(x)}) \\ a_2^{(t)} &= sim(h_t^{(dec)}, s_2^{(x)}) \\ &\vdots \\ a_L^{(t)} &= sim(h_t^{(dec)}, s_L^{(x)}) \end{aligned}$$

The similarity can be simply the inner product between vectors if they have same length

$$a_i^{(t)} = h_t^{(dec)} \cdot s_i^{(x)}$$

Now we can apply a softmax on the vector of all the similarities! The softmax turns every vector into scores in $[0, 1]$ that sum up to one!

$$\alpha^{(t)} = (\alpha_1^{(t)}, \ldots, \alpha_L^{(t)}) = softmax(a_1^{(t)}, \ldots, a_L^{(t)})$$

- Now we can use that $\alpha^{(t)}$ to compute c_t

$$c_t = \alpha_1^{(t)} s_1^{(x)} + \alpha_2^{(t)} s_2^{(x)} + \ldots + \alpha_L^{(t)} s_L^{(x)}$$

  The property of this is: the weight $\alpha_i^{(t)}$ is closer to one, if the updated state of the decoder is very similar to the encoder state $s_i^{(x)}$ at time step i. This is the attention.

- compute the softmax probabilities for the next output word o_t: this use the updated state $h_t^{(dec)}$ and the newly computed attention-weighted encoder states c_t

$$p(o_t|o_L, \ldots, o_{t-1}, x_1, \ldots, x_L) = softmax(W_{dec}[h_t^{(dec)}, c_t] + b)$$

  where $[,]$ denotes concatenation of vectors

Further reading on neural machine translation

A long tutorial:
Graham Neubig, *Neural Machine Translation and Sequence-to-sequence Models: A Tutorial*
`https://arxiv.org/pdf/1703.01619.pdf`

Outlook: other topics which combines image or video inputs

One can do more than machine translation with recurrent neural nets!

- video captioning `http://www.cv-foundation.org/openaccess/content_cvpr_2016/papers/Pan_Hierarchical_Recurrent_Neural_CVPR_2016_paper.pdf`,
- video story telling `https://sites.google.com/site/describingmovies/`.
- Visual question answering `https://arxiv.org/abs/1606.01847`, `https://arxiv.org/pdf/1612.00837.pdf`,