Ludwig-Maximilians-University

Department of Statistics

Sequence Modeling: Recurrent and Recursive Nets

Seminar

Markus Dumke

Munich, 19th December 2016

Abstract

RNN
r
r
r
r
r

Contents

1 Introduction

Recurrent Neural Networks (abbreviated RNNs) have been applied to various tasks involving sequential data, especially natural language processing. Applications include handwriting recognition and generation, machine translation, sentiment analysis, speech recognition, image captioning and many more.

In comparison to standard feed-forward neural networks, RNNs share parameters across time and are therefore much more efficient for sequential tasks as they can learn the structure of language, which does not change over time, while a feed-forward network must learn all rules of language for each position of time anew. For example a sentence might contain some time information (e.g. a date) and the information content is independent whether the date appears in the beginning or the end of the sentence.

RNNs can therefore overcome the two major drawbacks of feed-forward neural networks: the independence assumption and the fixed length assumption. They can process (in theory) arbitrary length sequences, one input at each time step, whilst feed-forward neural networks are dependent on a fixed-size input.

The hidden states can be seen as some kind of memory about the past and make RNNs very powerful learners, which are able to capture long-term dependencies.

In this paper we will explore basic RNN architectures and their applications, starting with Vanilla RNN, going through the optimisation and how to overcome the associated problems (especially the Vanishing Gradient Problem). Thereafter extensions of the basic RNN architecture like LSTM, Deep RNNs, Bidirectional RNNs, Recursive Neural Networks and Encoder-Decoder Networks will be covered. In the final part Google's Neural Machine Translation as a practical use-case will be explained.

2 Recurrent Neural Network

There exists a vast range of different Recurrent Neural Network architectures designed for different applications. Usually they have a hidden state h_t, which is shared over time, and inputs and outputs at one or more time steps. Recurrent Neural Networks can be seen as a dynamical system with the hidden state as a physical component, which is updated at each point in time. The same architecture can be unrolled over time showing the computational graph for each time step next to each other, compare Figure 1. In the following we will use the unrolled graph representation as this allows to look at which computations are performed in each time step.

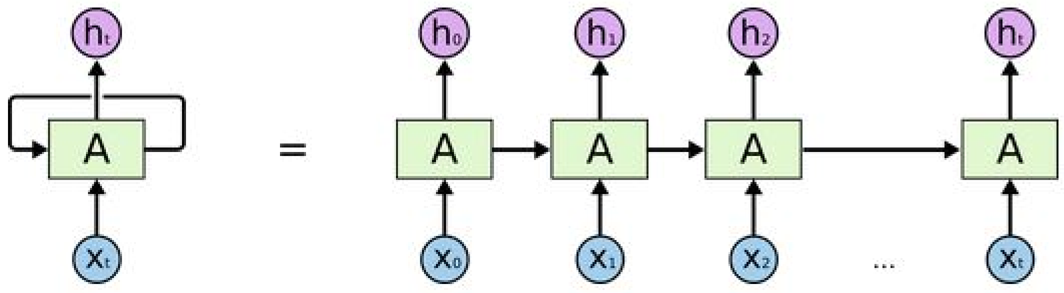

Figure 1: An RNN can be represented as a dynamical system or unrolled over time.
Source: Olah (2015b)

2.1 Vanilla RNN

The probably most basic Recurrent Neural Network (known as Vanilla RNN) consists of the following structure: At each time step an input $x^{(t)}$ is processed and an output, e.g. a probability distribution over words, is computed. The input could be a word or symbol represented as a vector, the output could be a probability distribution over all words/symbols, which can be used to sample the next word or symbol from the model. Such a model can be applied to language modelling, e.g. as a part of a chatbot. In the middle of the network are hidden states, which are fixed-sized vectors capturing relevant information from all previous inputs by learning a useful representation of the data. Recurrent connections between hidden states allow information from the past to be passed forward in time. Figure 2 shows the unrolled computational graph The RNN can be written as

$$\begin{aligned} a^{(t)} &= b + Wh^{(t-1)} + Ux^{(t)} \\ h^{(t)} &= tanh(a^{(t)}) \\ o^{(t)} &= c + Vh^{(t)} \\ \hat{y}^{(t)} &= softmax(o^{(t)}). \end{aligned} \tag{1}$$

The most commonly used activation function for Recurrent Neural Networks is the *tanh* function, which is symmetric around 0 with all values in the interval of $[-1, 1]$. If instead an non-zero mean activation function like the Sigmoid function would be used, the values of the hidden states would become greater over time, as each output of the Sigmoid is between $[0, 1]$. This results in an increasing bias.

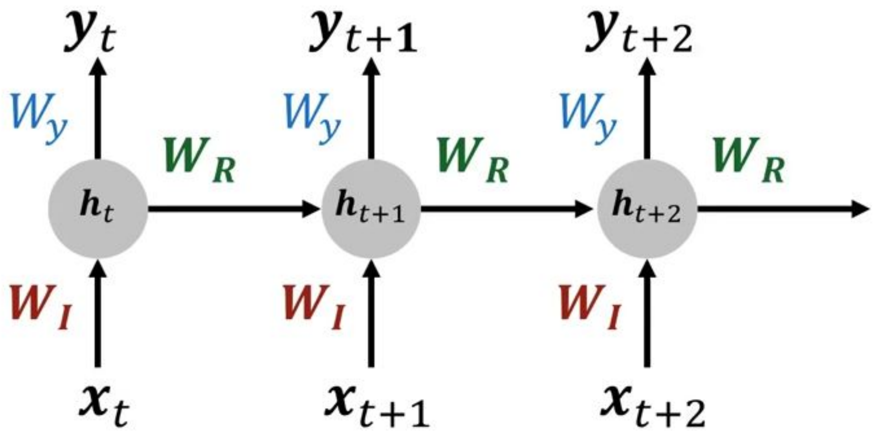

Figure 2: A Vanilla Recurrent Neural Network unrolled over time.
Source: Nervana (2016)

3 Optimisation and the Vanishing Gradient Problem

Recurrent Neural Networks can be trained with Backpropagation-through-time (BPTT), which is similar to the usual back-propagation algorithm used for training feed-forward neural networks, but contains connections over time. Usually minibatches of the training data are used for training. In the forward pass one minibatch (e.g. 32 symbols represented as vectors) is processed and moving forward in time each hidden state and output is computed according to Equation (3). For each time step the loss L_t between the predicted output $\hat{y}_t$ and the training target y_t is computed. The choice of loss function depends on the distribution of the targets, e.g. the cross-entropy loss for a categorical output distribution or more general the negative log-likelihood of the target distribution. The overall loss is then the sum of losses for each time step,

$$L(y, \hat{y}) = \sum_t L(y_t, \hat{y}_t) = -\sum_t logp(y_t|x_t) \tag{2}$$

In the backward pass the derivations of the final loss regarding all parameters are computed. Starting with the final time step we work our way backwards through time to obtain the gradient corresponding to the parameters using the chain rule of calculus. Then a Stochastic Gradient

Descent (SGD) update can be applied to all parameters:

$$\begin{aligned} W &= W - \epsilon \nabla_V L \\ U &= U - \epsilon \nabla_U L \\ V &= V - \epsilon \nabla_V L \\ b &= b - \epsilon \nabla_b L \\ c &= c - \epsilon \nabla_c L \end{aligned} \tag{3}$$

with learning rate ϵ.

As we are training on minibatches of training cases the computed gradient is a stochastic version of the true gradient obtained using all training cases. Each parameter is updated accordingly to its "‘share"’ of the final loss. SGD minimises the loss towards a local minimum.

3.1 The Vanishing Gradient Problem

The backpropagation of the gradient through a large number of time steps results in many multiplications of the same weights similar to the power method. If weights are smaller than 1, this leads to Vanishing Gradients else to exploding gradients, if weights are greater than 1. Both causes the RNN to fail learning long-term dependencies, so that only the very last time steps influence the current update. This is known as the Vanishing Gradient or Exploding Gradient problem. Whilst exploding gradients can be dealt with by Gradient Clipping, vanishing gradients are a much more sincere problem and special architectures are needed to overcome the problem. The most famous architecture is the Long-Short-Term-Memory (LSTM) which replaces the simple computation of the hidden state by a much more complex computation involving several equations.

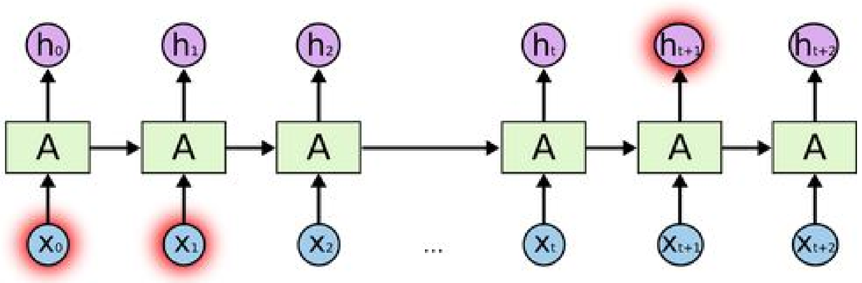

Figure 3: If the distance in time gets large a Vanilla RNN fails to learn the dependency on inputs that are many time steps away due to the Vanishing Gradient Problem.
Source: Olah (2015b)

3.2 Solutions to Optimisation Difficulties

As already mentioned Gradient Clipping provides a simple and often used trick to prevent the Explosion of Gradients. Therefore the norm of the Gradient is clipped if its exceeds a certain threshold.

Another solution is to use a special regularisation term, which ensures that information can flow over long time ranges (Goodfellow et al. (2016)). The gradient of the loss with respect to a hidden state h_t should be of similar size to the product of this gradient with the derivation of the hidden state to the previous hidden state h_{t-1}:

$$\nabla_{h_t} L \approx (\nabla h_t L) \frac{\partial h_t}{\partial h_{t-1}} \tag{4}$$

Therefore the following regularisation term can be added to the loss function:

$$\Omega = \sum_t \left(\frac{||(\nabla_{h_t} L) \frac{\partial h_t}{\partial h_{t-1}}||}{||\nabla_{h_t} L||} - 1 \right)^2 \tag{5}$$

Another idea is to add skip connections through time, which enable the gradient to flow back longer time ranges and learn dependencies from further back in time.

4 LSTM

The LSTM provides a different way of dealing with Vanishing Gradients. The simple architecture described in Section 2 is now changed. A cell state s_t is introduced, which can be manipulated by different gates to forget old information, add new information and read information out of the cell state. Each gate is a vector of the same size as the cell state and each element of the vector is between 0 and 1, with 0 meaning "‘let nothing through"’ and 1 "‘let everything through"’. Each gate is computed as a function of the previous hidden state h_{t-1} and the input at the current time step x_t multiplied by weights and squashed through a Sigmoid function into the range of $[0, 1]$.

$$LSTM formulas \tag{6}$$

There are lots of variants of the described LSTM structure, including connections from the cell state to the gates as another input or combining different gates into one, e.g. the forget gate and input gate into one update gate.

4.1 Gated Recurrent Unit and variants

The Gated Recurrent Unit (GRU) is an often used variant of the LSTM with a simpler structure. Instead of a forget, input and output gate there are only an update gate and a reset gate. There is no cell state s_t, instead the hidden state h_t is directly manipulated.

$$h_t = u_t h_{t-1} + (1 - u_t)\sigma\left(b + Ux_{t-1} + Wr_t h_{t-1}\right) \tag{7}$$

Both LSTM and GRU achieve similar performance on many tasks

5 Application 1: LSTM

5.1 Sampling from an RNN

As a generative model an RNN allows us to sample from the conditional distributions at each time step. This is especially useful, if we want to generate sequences ourselves, e.g. texts, programming code, handwriting or speech.
A specialty of sequential data is to determine the length of the sequence. A network without an input parameter determining the length of the sequence would perform poorly as the sequence could stop in the middle of a sentence. There are several ways to control the sequence length:

- special end symbol
- Bernoulli random variable
- integer value giving the length

If we want to generate sequences from a dictionary we could create a special end token, which ends the sequence when its sampled. Another more general way would be to sample an additional Bernoulli output at each time step, determining whether to continue or stop the sequence. A third way would be to sample an integer value determining the length of the sequence. Therefore we need to give the network an additional input at each time step regarding of how close it is to the end of episode. In this way we can avoid sequences that stop in the middle of a sentence. Some interesting applications of RNNs to generate text are given by ...

6 Extensions of Recurrent Neural Network

6.1 Recurrent Connections from output to hidden units

RNNs that have only recurrent connections between the output at one time step to the hidden state at the next time step are a special kind of RNN. They are less powerful because the only information carried on from the past is the actual output of the previous time step, which is trained to match the training labels and therefore cannot carry arbitrary information about the whole past sequence as is possible with recurrence between hidden units. The upside of this is that they are much easier to train as each time step can be decoupled and therefore parallelisation speeding up the training is possible.
The training algorithm for such output-to-hidden RNNs is called **Teacher forcing**. Whilst training the true outputs y from the training data are used as an input in the next time step instead of the model-generated outputs, because those are trained to be close to the true outputs. Therefore each time step is independent of another and can be trained in parallel.
At test time the true outputs y are usually not available so the self-generated outputs are fed back into the sytem. A problem that can arise from this is that inputs at train and test time can vary greatly in size. Possible solutions include training on both teacher-forced and free-running inputs and increase as kind of a curriculum learning strategy the ratio of generated values to real data.

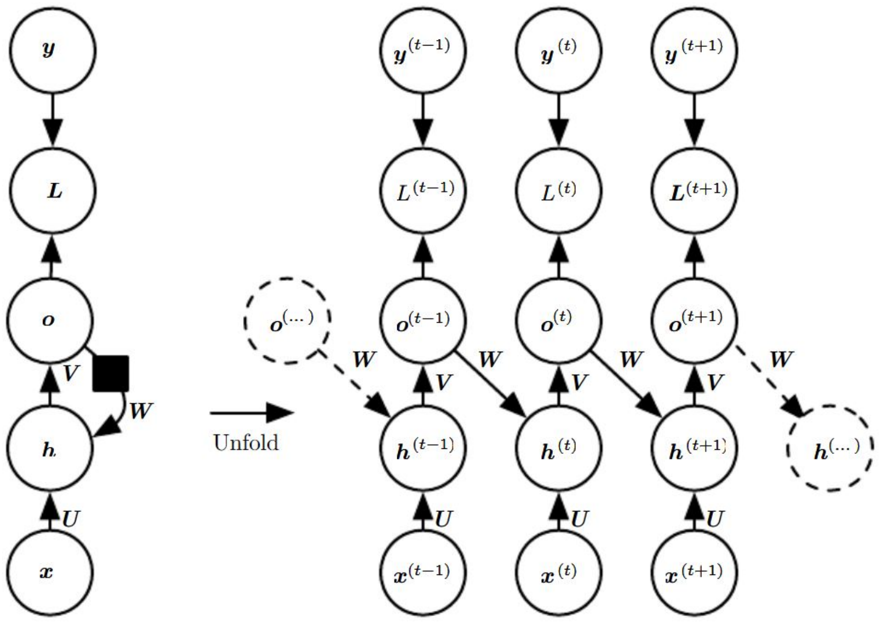

Figure 4: A Recurrent Neural Network with recurrent connections from the output to the hidden units.
Source:

6.2 One-output RNN

Recurrent Neural Networks do not need to have an output at each time step, instead they can only have outputs at a few time steps. A common variant is an RNN with only one output at the end of the sequence. Information from the whole input sequence is incorporated into the final hidden state, which is then used to create an output, e.g. a sentiment ("‘positive"’, "‘neutral"’ or "‘negative"’) for a movie review, see Figure 5. Other applications of such an architecture are sequence labeling, e.g. classify an article into different categories ("‘sports"’, "‘politics"’ etc.).

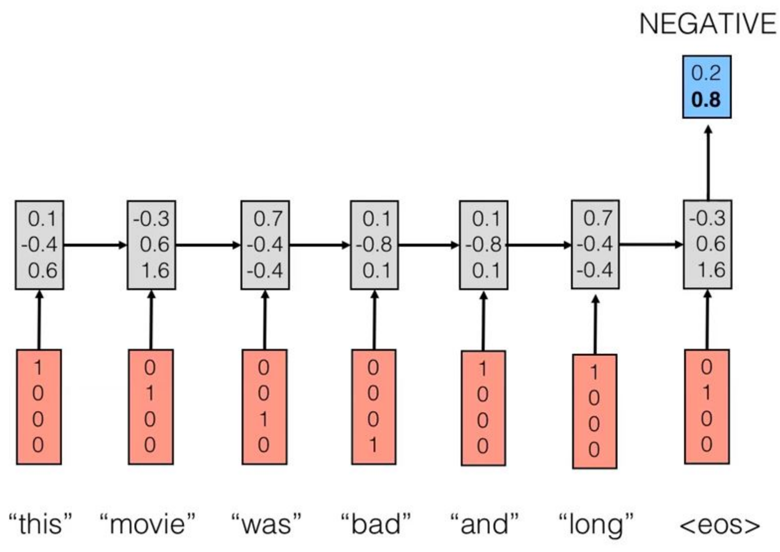

Figure 5: A Recurrent Neural Network with one output at the end of the sequence. Such a model can be used for sentiment analysis.
Source: Nervana (2016)

6.3 RNN based on context

There are multiple ways to include additional inputs that do not change over time in an RNN. The information could be incorporated into the hidden state h_0 or given as a bias term in each time step.

6.4 Deep RNNs

RNNs can be made deep in all computations that are performed. One way to do this is to add multiple layers of hidden units in hierarchical groups, another way is to introduce deep computations between hidden to hidden units, input to hidden units or hidden to output units. Instead of a simple nonlinearity as the tanh function, a feed-forward neural network (MLP) is used to compute the next unit. This can be combined with skip connections, which skips the current MLP calculation and enables faster flow of information through the layers. Deep

RNNs are very powerful but have the disadvantage of a vast number of parameters and therefore difficult training.

6.5 Bidirectional RNN

Another generalisation of the RNN structure are bidirectional RNNs. Those allow to process sequential data depending on both the past and future inputs, e.g. an application predicting missing words, which probably depend on both the preceding words and the following words. Bidirectional RNNs are also used for the encoding of a sequence in Machine Translation.

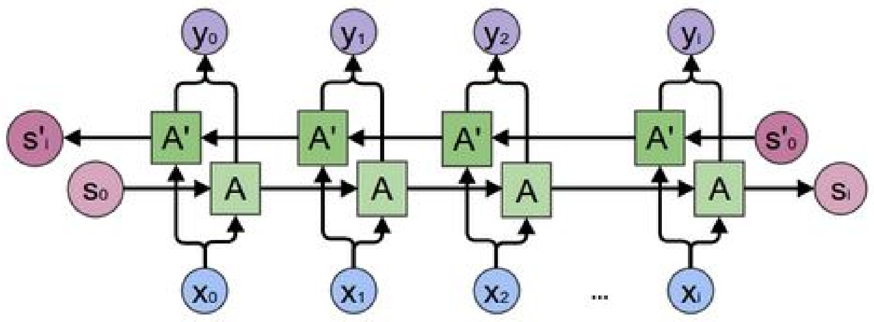

Figure 6: A bidirectional RNN
Source: Olah (2015a)

6.6 Recursive Neural Networks

Recursive Neural Networks are a generalisation of Recurrent Neural Networks. A tree structure instead of a chain structure is used for the computations of the RNN.

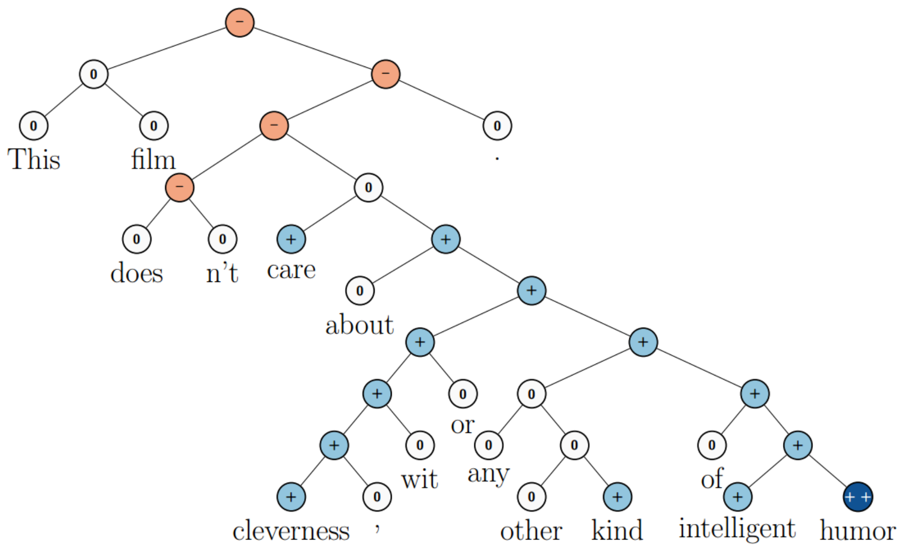

Figure 7: A recursive neural network
Source:

6.7 Encoder-Decoder Architecture

The encoder-decoder (also called sequence-sequence) framework allows us to generalise the idea of an RNN to different-length input sequences and output sequences. An input/encoder-RNN processes the input of length n_x and omits a typically fixed-length vector C, the context, usually a function of the last hidden state of the encoder-RNN. The output/decoder-RNN processes this context and outputs a sequence of length n_y, which could vary from n_x. Such an architecture can be used for machine translation or text summaries. Another application is image captioning, where the encoder RNN is typically a Convolutional Neural Network, which extracts relevant features from the image, on which the decoder RNN can focus and generate a description.

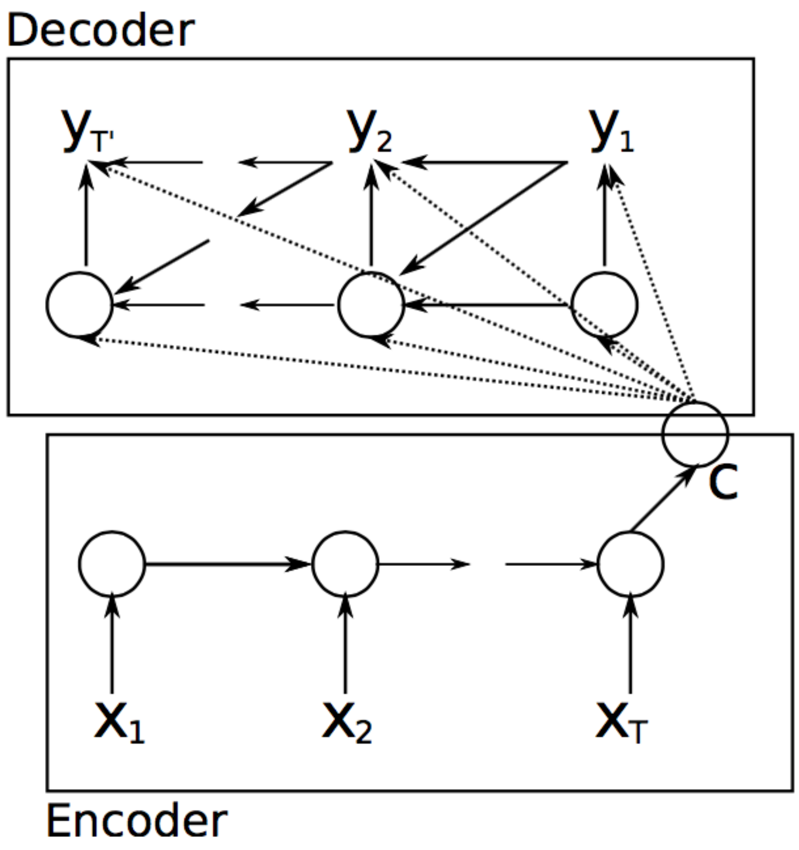

Figure 8: Encoder decoder
Source:

6.8 Attention

Bahdanau et al. (2015) introduced the concept of attention to Neural Machine Translation. In a classical decoder-encoder RNN all information about the input sequence must be incorporated into the final hidden state h_t, which is then passed as an input to the decoder network. With a long input sequence the fixed-sized hidden state is unlikely to capture all relevant information about the past and therefore learning is poor. An attention mechanism allows the decoder

network now to focus on different parts of the input sequence by adding connections from all hidden states to the output RNN. The attention mechanism can be seen as a set of weights, which allow the network to focus on relevant information from the input sequence
Luong et al. (2015) distinguish between a global and local attention mechanism. The global attention model exhibits connections to all hidden states h_1 to h_t of the input RNN, the attention distribution is therefore a variable-length vector of weights. The context vector c_t is then a weighted average of

$$c_t = \sum_t a_t h_t. \tag{8}$$

In comparison a local attention mechanism only focuses on a subset of the hidden states centered around an alignment position p_t to predict the current target y_t. The context c_t is a weighted average of the hidden states, which are inside the alignment window.

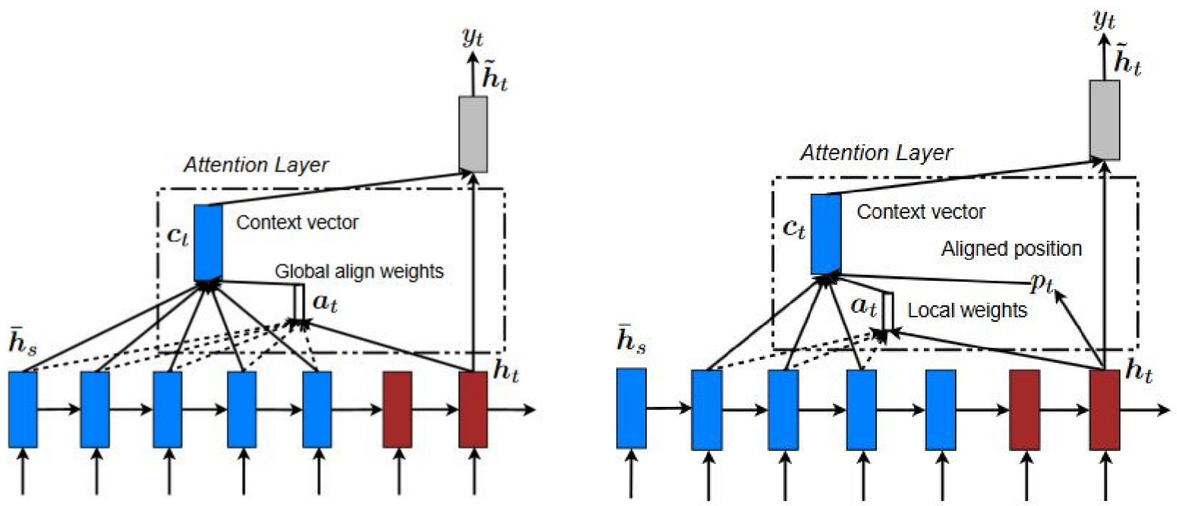

Figure 9: Attention mechanism, global on the left, local on the right
Source: Luong et al. (2015)

Attention works very well for different tasks such as Machine Translation or Image Captioning by allowing the model to look at relevant parts of the input, e.g. a specific part of the image, for which then a text description is generated.

6.9 Res-Net

Residual Networks (called Res-Nets) He et al. (2015) are useful when training deep neural network architectures. It is similar to dropout, which samples at each time step if a unit is included in the model and creating an ensemble of models. Instead of crossing out single units residual networks operate on whole layers. This can be seen as skip connections through time.

7 Application 2: Google's Neural Machine Translation

Google's Neural Machine Translation (Wu et al. (2016)) is an application of a Recurrent Neural Network to the task of machine translation acieving state-of-the-art performance.
The architecture consists of an encoder-decoder architecture each build of several stacked LSTMs. In the encoder the first two hidden layers are processed by a bidirectional RNN, thereafter LSTMs with forward flow of information are stacked. For training this deep architecture residual connections between layers are used. An attention mechanism allows the decoder RNN to focus on different parts of the hidden states from the encoder RNN. In the decoder the translation of the input sequence is done with multiple stacked LSTMs (8 layers) and residual connections starting in the second layer. In the final layer a softmax is applied to get a probability distribution over words.

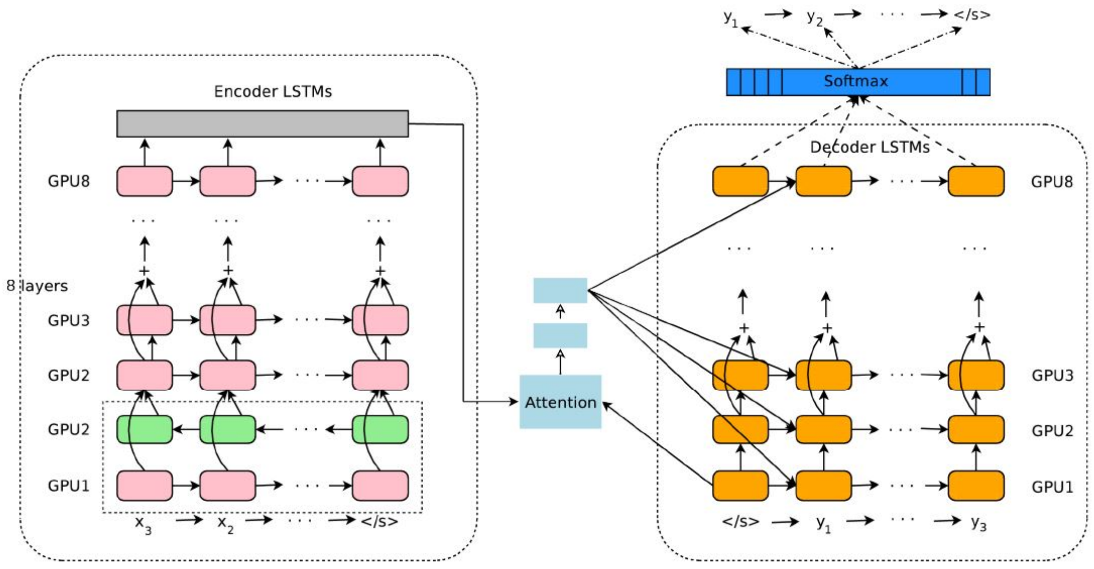

Figure 10: Neural Machine Translation
Source: Wu et al. (2016)

In November 2016 Google announced that the Neural Machine Translation is now capable of zero-shot machine translation, which is translating between language pairs which were not specifically trained. The model often finds a vector representation of sentences (called "‘interlingua"’), where in the meaning similar sentences are clustered together across different languages.

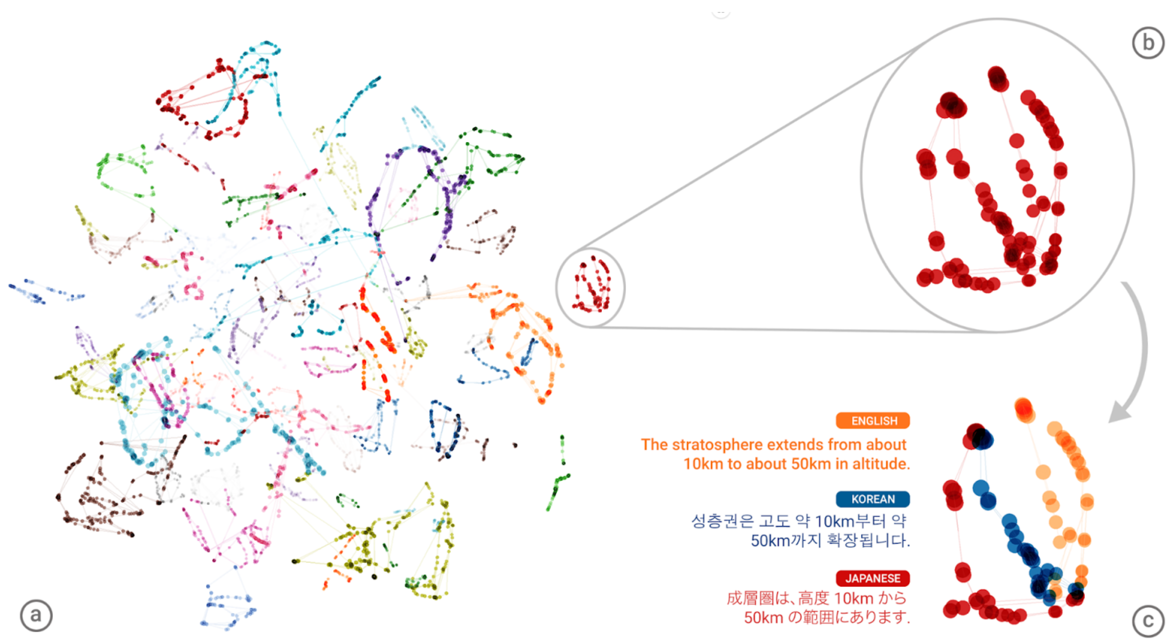

Figure 11: Neural Machine Translation language embeddings
Source: Johnson et al. (2016)

References

Goodfellow, I., Bengio, Y. and Courville, A. (2016). Deep learning. Book in preparation for MIT Press.
URL: *http://www.deeplearningbook.org*

He, K., Zhang, X., Ren, S. and Sun, J. (2015). Deep residual learning for image recognition, *CoRR* **abs/1512.03385**.
URL: *http://arxiv.org/abs/1512.03385*

Johnson, M., Schuster, M., Le, Q. V., Krikun, M., Wu, Y., Chen, Z., Thorat, N., Viégas, F. B., Wattenberg, M., Corrado, G., Hughes, M. and Dean, J. (2016). Google's multilingual neural machine translation system: Enabling zero-shot translation, *CoRR* **abs/1611.04558**.
URL: *http://arxiv.org/abs/1611.04558*

Luong, M., Pham, H. and Manning, C. D. (2015). Effective approaches to attention-based neural machine translation, *CoRR* **abs/1508.04025**.
URL: *http://arxiv.org/abs/1508.04025*

Nervana (2016). Understanding lstm networks, https://www.nervanasys.com/recurrent-neural-networks.

Olah, C. (2015a). Neural networks, types, and functional programming, http://colah.github.io/posts/2015-09-NN-Types-FP/.

Olah, C. (2015b). Understanding lstm networks, http://colah.github.io/posts/2015-08-Understanding-LSTMs/.

Wu, Y., Schuster, M., Chen, Z., Le, Q. V., Norouzi, M., Macherey, W., Krikun, M., Cao, Y., Gao, Q., Macherey, K., Klingner, J., Shah, A., Johnson, M., Liu, X., Kaiser, L., Gouws, S., Kato, Y., Kudo, T., Kazawa, H., Stevens, K., Kurian, G., Patil, N., Wang, W., Young, C., Smith, J., Riesa, J., Rudnick, A., Vinyals, O., Corrado, G., Hughes, M. and Dean, J. (2016). Google's neural machine translation system: Bridging the gap between human and machine translation, *CoRR* **abs/1609.08144**.
URL: *http://arxiv.org/abs/1609.08144*

List of Figures