Ludwig-Maximilians-University

Department of Statistics

Sequence Modeling: Recurrent and Recursive Nets

Seminar

Markus Dumke

Munich, 9th January 2017

Abstract

In the last years deep learning has received lots of attention and many applications and models were designed. One especially useful architecture are Recurrent Neural Networks (RNNs). RNNs are neural networks suited for sequential data. In particular Recurrent Neural Networks have been applied to language tasks such as speech recognition and generation, machine translation, image captioning and other applications which involve processing sequences. This paper will review different recurrent neural network architectures and present the challenges when optimizing such models. In the last part we will have a look at Google's Neural Machine Translation, recently introduced by (Wu et al. (2016)) and now deployed by Google Translate service.

Contents

1 Introduction

Recurrent Neural Networks (abbreviated RNNs) are statistical models, which can process sequences of data. 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 two major drawbacks of feed-forward neural networks: the independence assumption and the fixed length assumption. With feed-forward neural networks each input will be processed using new input-specific weights, whilst in RNN we will use the same weights for each time step again. Statistical strength is shared over the model and as we have a fixed set of parameters RNNs can process much longer sequences than feed-forward networks.

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

In recent years Recurrent Neural Networks have been applied to various tasks involving sequential data, especially in the field of natural language processing. Applications include handwriting recognition and generation, machine translation, sentiment analysis, speech recognition and generation, image captioning and many more.

In this paper we will explore different RNN architectures and their applications, starting with Vanilla RNNs, going through the optimization and how to overcome the associated problems (especially the vanishing gradient problem). Thereafter extensions of the basic RNN architecture like the LSTM, GRU, deep RNNs, residual networks, bidirectional RNNs, recursive neural networks, encoder-Decoder Networks and the attention mechanism 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 one or more hidden states h_t for each time point, and inputs and outputs at one or more time steps. The hidden states of the network are connected. Each hidden state h_t is a function of a varying number of preceding inputs $x_1, ..., x_t$, but in an RNN this can be parametrized as a transition from one state to the next:

$$\begin{aligned} h_t &= g_t(x_t, x_{t-1}, ..., x_1) \\ &= f(h_{t-1}, x_t) \end{aligned} \tag{1}$$

This way the input has the same fixed-sized dimension and the same transition function f can be applied each time step.

Recurrent Neural Networks can be seen as a dynamical system with the hidden state as a physical component like a biological cell, which is "updated" at each point in time. The same architecture can also be unrolled over time showing the computational graph each time step next to each other, compare Figure 1. In the following we will use the unrolled graph representation, which shows us, which computations are performed in each time step and is therefore useful to implement such a model.

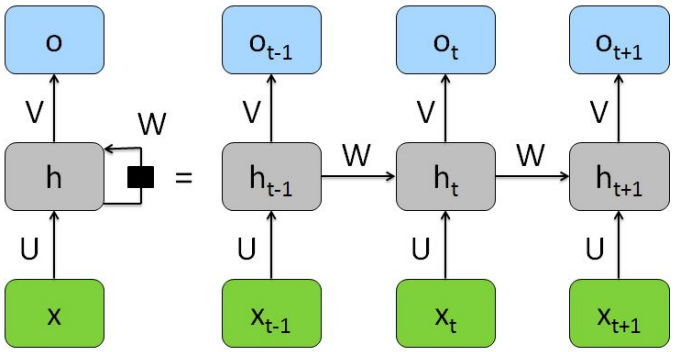


Figure 1: An RNN can be represented as a dynamical system or unrolled over time. The black square denotes a delay of one time step.

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 o_t 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 modeling, 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. The right part of Figure 1 shows the unrolled computational graph. An RNN for categorical

outputs can be written as

$$
\begin{aligned}
a_t &= b + W\, h_{t-1} + U\, x_t \\
h_t &= \tanh(a_t) \\
o_t &= c + V\, h_t \\
\hat{y}_t &= \text{softmax}(o_t),
\end{aligned}
\tag{2}
$$

where a_t gives the activation for the hidden state h_t and o_t is a vector of unnormalized log-probabilities for each output category. $\hat{y}_t$ then defines a probability distribution over the output categories.

Note that parameters are shared over time, so at each time step the same activation function, weights W, U, V and biases b, c are used. This corresponds to a stationarity assumption. The most commonly used activation function for Recurrent Neural Networks is the hyperbolic tangent (tanh) function, which is symmetric around 0 with all values in the interval of $[-1, 1]$.

2.1 Language model

A simple character-level language model could look like this:

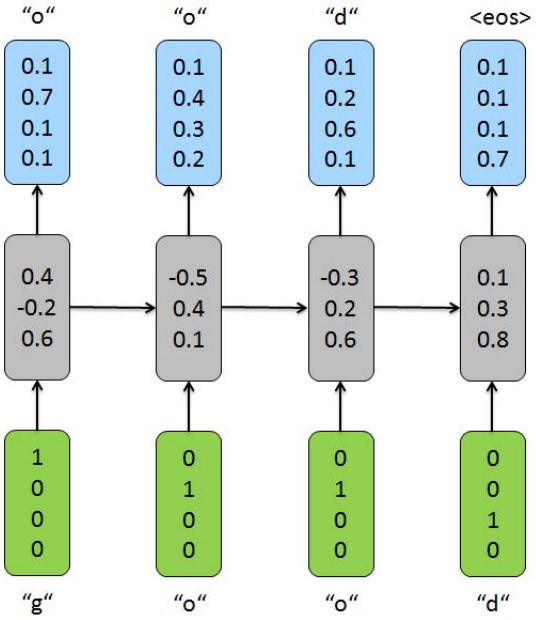


Figure 2: A simple application of a vanilla RNN for language modeling.

At each time point an input vector x_t representing a single character in one-hot-coding is processed, a single hidden state, a vector h_t, is computed and an output $\hat{y}_t$, a probability distribution over all characters is omitted. With a conditional independence assumption

$$
P(y_1, y_2, ..., y_n) = \prod_{i=1}^{n} P(y_i \,|\, y_1, ..., y_{i-1}) \tag{3}
$$

the next character can be sampled based on the history of all previous characters and the probability of observing a word or sentence can be computed.

2.2 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 added to the dictionary
- 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. This output is trained to predict correctly, if the sequence ends or not. 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. This way we can avoid sequences that stop in the middle of a sentence.
At test time a simple beam search over possible sentences could be performed.

3 Optimization 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 the training data is split into minibatches. It can be useful to have sentences of similar length in one minibatch (Sutskever et al. (2014)). Before training all model parameters should be initialized to small random numbers.
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 Equations (2). 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 mean-squared error (MSE) for a regression problem or the cross-entropy loss for a categorical output distribution. As with feed-forward neural networks a wide variety of loss functions can be used. Usually we try to minimize the negative log-likelihood of the target distribution. The overall loss is then the sum of losses for each time step,

$$L(y, \hat{y}) = \sum_t L(y_t, \hat{y}_t) = -\sum_t \log\, p(y_t|x_t) \tag{4}$$

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:

$$W = W - \epsilon \nabla_W L \tag{5}$$

and similar updates to all other parameters U, V, b, c with learning rate ϵ. The learning rate should be decreased over time.
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.

3.1 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, large weights to exploding gradients. 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. While 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 more complex computation involving several equations, compare Section 4.

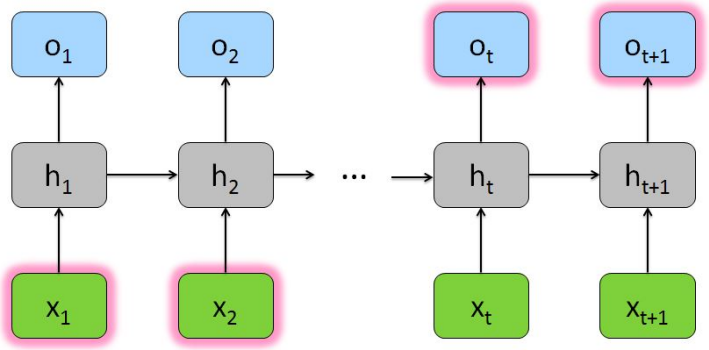


Figure 3: If the distance in time gets large a vanilla RNN fails to learn the dependency on inputs that are many time steps away due to the vanishing gradient problem.

3.2 Solutions to Optimization Difficulties

Gradient clipping provides a simple and often used trick to prevent the explosion of gradients. There are different ways to do this. One solution is to clip the norm of the gradient, if it exceeds a certain threshold and shrink the gradient equally for all parameters. Another way is to simply clip all elements of a parameter vector or matrix, which are greater than an upper limit (Goodfellow et al. (2016)).

To prevent vanishing gradients a special regularization term can be used, 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}:

$$\nabla_{h_t} L \approx (\nabla h_t L) \frac{\partial h_t}{\partial h_{t-1}} \tag{6}$$

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

$$\Omega = \sum_t \left(\frac{||(\nabla_{h_t} L) \frac{\partial h_t}{\partial h_{t-1}}||}{||\nabla_{h_t} L||} - 1 \right)^2 \tag{7}$$

Another idea is to add skip connections through time, which enable the gradient to flow back longer time ranges and therefore learn dependencies from further back in time. The network now operates on multiple time-scales. This is essentially what is done in residual networks (Section 5.5). Leaky units are another way to allow information to persist. Hidden states are connected with a linear self-connection similar to a running average (Goodfellow et al. (2016)):

$$\mu = \alpha \, \mu_{t-1} + (1 - \alpha) \, \nu_t \tag{8}$$

Short-term connections can also be entirely removed, forcing the units to focus on long-term interactions.

4 Gated RNNs

4.1 Long-Short-Term-Memory (LSTM)

The LSTM provides a different way of dealing with vanishing gradients. The simple architecture described in Section 2 is now modified. 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 it. Each gate is a vector of the same size as the cell state and each element of the vector is a number between 0 and 1, with 0 meaning "let nothing through" and 1 "let everything through". The gates are computed as a parametrized function of the previous hidden state h_{t-1} and the input at the current time step x_t multiplied by gate-specific weights and squashed through a Sigmoid function into the range of $[0, 1]$. The cell state allows the recurrent neural network to

keep information over long time ranges and therefore overcome the vanishing gradient problem. The equations for the three gates are

$$\begin{aligned} f_t &= \sigma\left(b^f + U^f\, x_t + W^f\, h_{t-1}\right) \\ i_t &= \sigma\left(b^i + U^i\, x_t + W^i\, h_{t-1}\right) \\ q_t &= \sigma\left(b^q + U^q\, x_t + W^q\, h_{t-1}\right). \end{aligned} \tag{9}$$

The cell state s_t is computed as a function of the previous cell state s_{t-1} multiplied by the forget gate f_t, which tells us how much of the old information is discarded. New information is added as a function of the input x_t and the previous hidden state h_{t-1} scaled by the input gate i_t. The hidden state of the network is then a function of the LSTM cell state multiplied by the output gate.

$$\begin{aligned} s_t &= f_t\, s_{t-1} + i_t \tanh\left(b + U\, x_t + W\, h_{t-1}\right) \\ h_t &= \tanh(s_t)\, q_t \end{aligned} \tag{10}$$

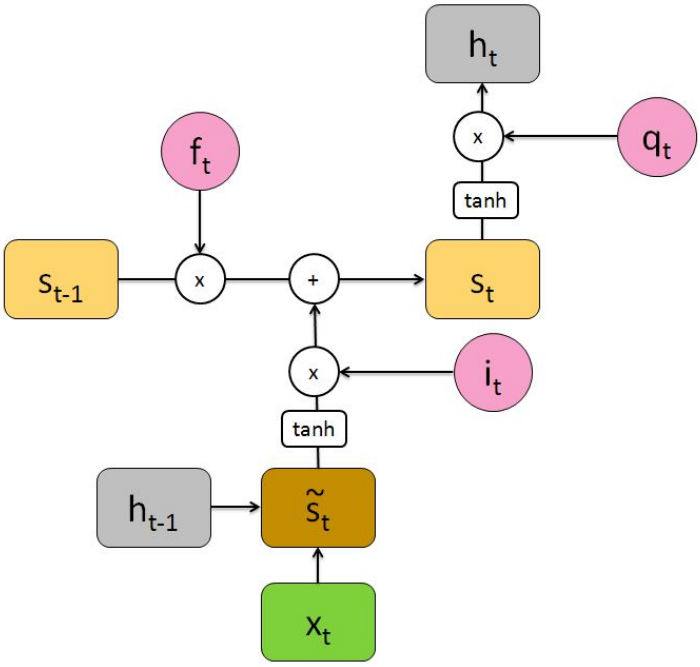


Figure 4: The LSTM architecture is used to model long-term-dependencies.

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

4.2 Gated Recurrent Unit (GRU)

The Gated Recurrent Unit (GRU) is an often used variant of the LSTM with a simpler structure (Cho et al. (2014)). 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, compare Figure 5.

$$h_t = u_t\, h_{t-1} + (1 - u_t)\ \tanh\left(b + U\, x_{t-1} + W\, r_t\, h_{t-1}\right) \tag{11}$$

The gates are computed similar to the LSTM as a sigmoid non-linearity,

$$\begin{aligned} r_t &= \sigma\left(b^r + U^r\, x_t + W^r\, h_{t-1}\right) \\ u_t &= \sigma\left(b^u + U^u\, x_t + W^u\, h_{t-1}\right). \end{aligned} \tag{12}$$

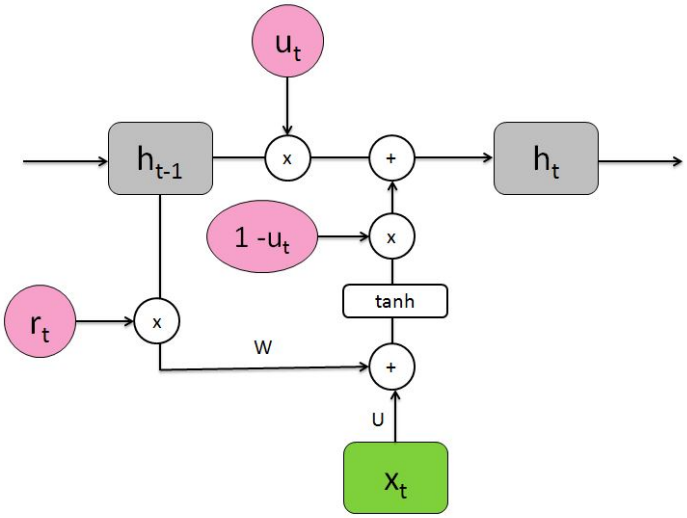


Figure 5: The GRU architecture.

Jozefowicz et al. (2015) compared many different gated RNN architectures including LSTM and GRU variants. They found that the GRU outperforms the LSTM on all tasks except language modeling and that an LSTM with dropout achieves the highest performance in the latter task. They recommend to add a bias of 1 to the LSTM forget gate, which is essential to the LSTM's success.

5 Extensions of Recurrent Neural Networks

5.1 Recurrent connections from output to hidden units

Recurrent Neural Networks that have only recurrent connections between the output at one time step and the hidden state at the next time step are a special kind of RNN. They are less powerful than those networks including recurrence between hidden units, 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. Therefore it is much more restraint about the information content it can pass on in time as is possible with recurrence between hidden units. An advantage thereof is that they are much easier to train as each time step can be decoupled and therefore parallelization speeding up the training is possible.

The training algorithm for such output-to-hidden RNNs is called **Teacher forcing**. While training, model-generated outputs o_t can be replaced by the true labels y_t from the training data, as those give the ground truth of what we train the model-generated outputs to be close to. 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.

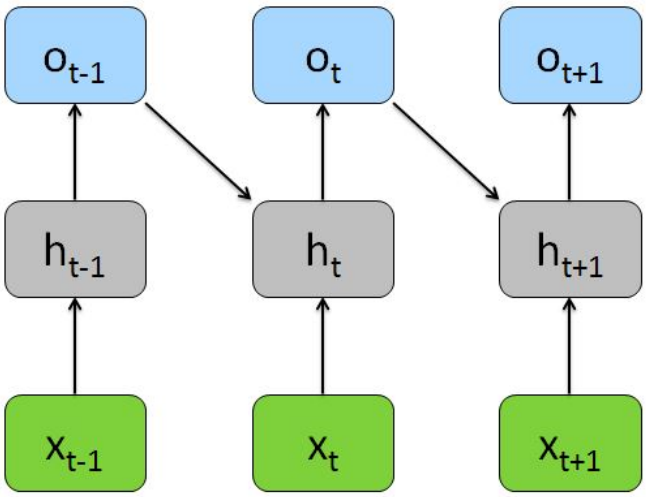


Figure 6: A Recurrent Neural Network with recurrent connections from the output to the hidden units.

5.2 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 7. Other applications of such an architecture are sequence labeling, e.g. classify an article into different categories ("sports", "politics" etc.).

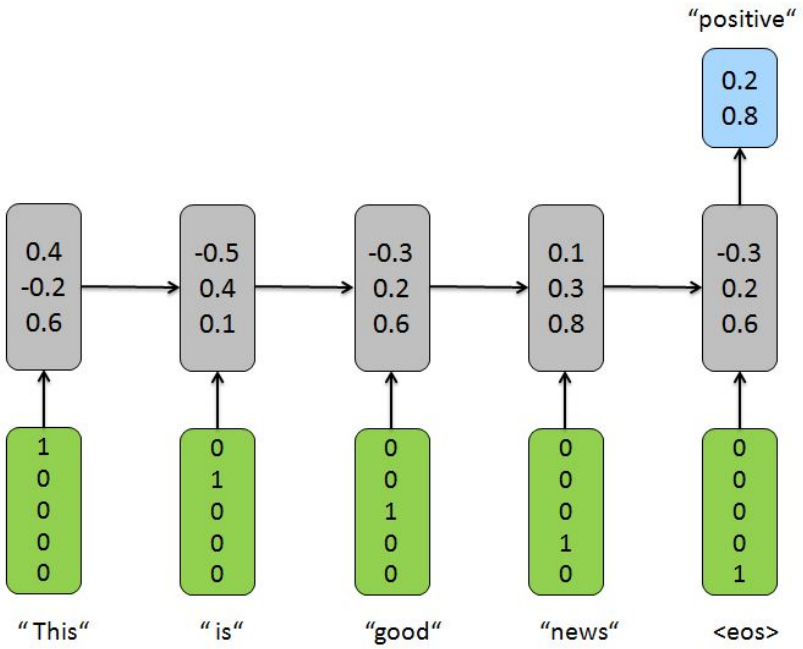


Figure 7: A Recurrent Neural Network with one output at the end of the sequence. Such a model can be used for sentiment analysis.

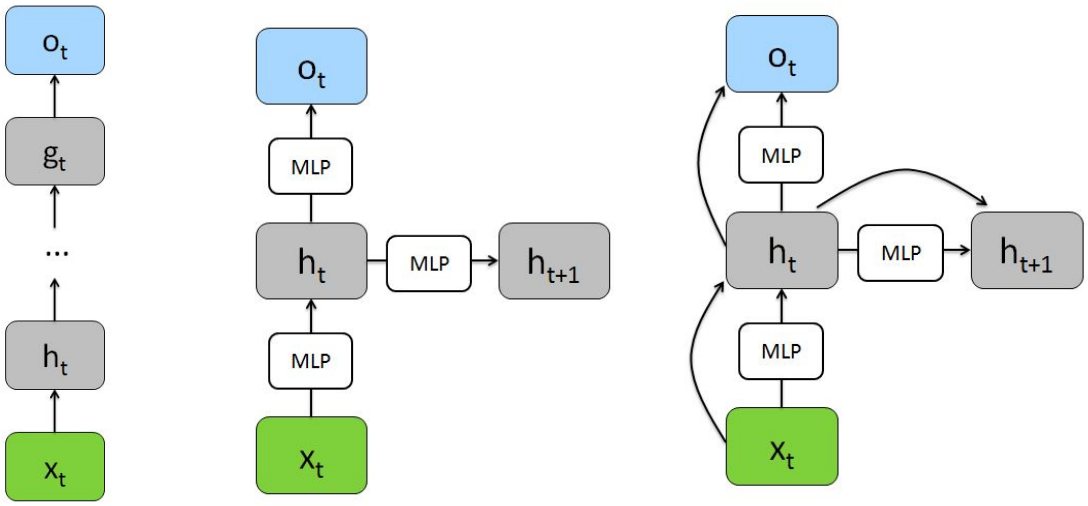
5.3 RNN based on context

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

5.4 Deep RNNs

Recurrent neural networks can be made deep in different ways (Goodfellow et al. (2016)). 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 non-linearity 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 and can achieve better performance than shallow architectures, but are much more prone to the vanishing gradient problem and therefore training is more difficult.



(a) Stacked hidden layers. **(b)** Feed-forward neural networks are used to compute the next state. **(c)** Skip connections allow faster training.

Figure 8: There are several ways to make a recurrent neural network deep.

5.5 Residual Network

Residual Networks (called Res-Nets) are a kind of deep neural network, which can have more than 100 or even 1000 layers and is therefore much deeper than previously used architectures (He et al. (2015)). As usual each layer is connected to the next layer by a linear transformation followed by a non-linearity. Special about residual networks are identity skip connections, connecting each layer with any subsequent layer thereby skipping all layers in between. With these skip connections short paths through the otherwise deep network are created, which the gradient can flow along. This circumvents the vanishing gradient problem even in very deep networks. While training this can be seen as an ensemble of relatively shallow architectures. Veit et al. (2016) found that in a 110 layer-network only paths with depth of 10-34 layers influenced the computation of the gradient. This way it is similar to dropout, but instead of sampling single units residual networks sample whole layers to be included in a path or not.
Huang et al. (2016) introduce the concept of stochastic depth to residual networks. While training complete layers are randomly skipped, reducing the overall depth of the network. For each minibatch a random subset of layers is selected and dropped from the model. This technique differs from dropout, which drops single units and therefore makes a network thinner. Stochastic depth allows the model to train an implicit ensemble of shallow networks and at test time these are combined to a deep network.
Residual networks can be used with all kinds of neural network architectures, with feed-forward

networks as well as with recurrent networks, e.g. in a stacked LSTM model.

5.6 Bidirectional RNN

Another generalization of the simple RNN described in Section 2 are bidirectional RNNs. These allow to process sequential data depending on both past and future inputs, e.g. an application predicting missing words, which probably depend on both preceding and following words. One RNN processes inputs in the forward direction from x_1 to x_T computing a sequence of hidden states $(h_1, ..., h_T)$, another RNN in the backward direction from x_T to x_1 computing hidden states $(g_T, ..., g_1)$. Predictions are then based on both hidden states, which could be concatenated. With connections going back in time the whole input sequence must be known in advance to train and infer from the model. Bidirectional RNNs are often used for the encoding of a sequence in machine translation, e.g. by Bahdanau et al. (2014). This allows the RNN to encode more of the information in the hidden states of the network. For example two related words might be split up, so they are found in different parts of the input sentence, but should be translated as a single word. This is a case, where an RNN can improve translation performance. See also Sections 5.8 and 5.9.
Bidirectional RNNs are also useful for speech recognition, where the meaning of a word may depend on preceding and directly following phonemes.

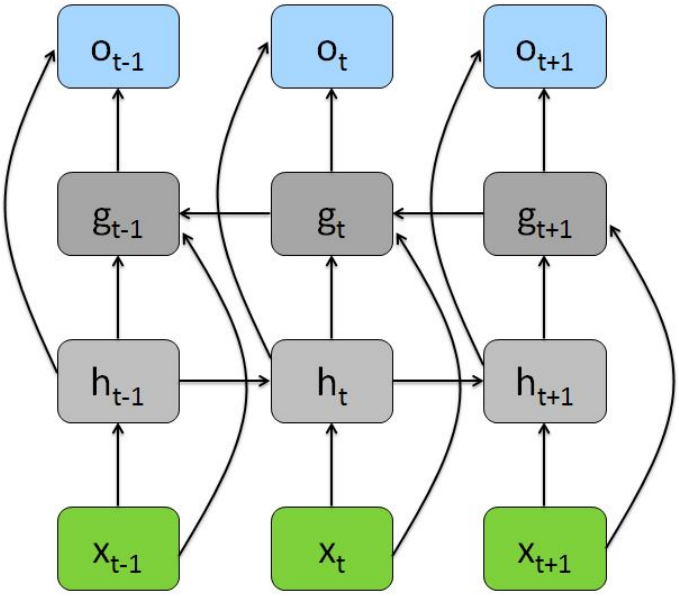


Figure 9: A bidirectional RNN consists of a forward RNN processing inputs from left to right and a backward RNN processing inputs backwards in time.

5.7 Recursive Neural Network

Recursive Neural Networks are a generalization of Recurrent Neural Networks. A tree structure instead of a chain structure is used for the computations of the RNN. A fixed set of weights is

repeatedly applied to the nodes of the tree. Recursive neural networks have been successfully applied to sentiment analysis (Socher et al. (2013)).

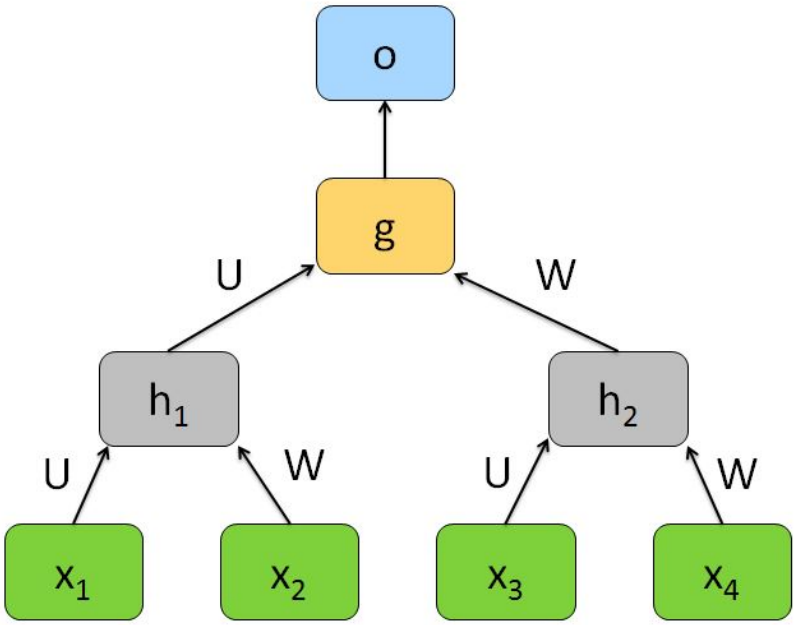


Figure 10: A recursive neural network

5.8 Encoder-Decoder Network

Standard RNNs operate on input and output sequences of the same length. But for many interesting tasks such as question answering or machine translation the network needs to map an input sequence to an output sequence of different length. This is what an encoder-decoder (also called sequence-to-sequence architecture) enables us to do. An input/encoder-RNN processes the input sequence of length n_x and omits a fixed-length context vector C, usually the final hidden state or a simple function thereof. One time step after the other information from the input sequence is processed, added to the hidden state and passed forward in time through the recurrent connections between hidden states in the encoder. The context summarizes important information from the input sequence, e.g. the intent of a question in an question answering task or the meaning of a text in the case of machine translation. This context is provided as an input to the decoder/output-RNN at each point in time. The decoder RNN uses this information to predict the output, a sequence of length n_y, which could vary from n_x. In natural language processing the decoder is a language model with recurrent connections between the output at one time step and the hidden state at the next time step as well as recurrent connections between the hidden states. The model is now

$$p(y_1, ..., y_{n_y} | x_1, ..., x_{n_x}) = \prod_{t=1}^{n_y} p(y_t | C; y_1, ..., y_{t-1}), \tag{13}$$

where C is the context vector. This architecture is now jointly trained to minimize the translation error given a source sentence. Each conditional probability is then

$$p(y_t|y_1, ..., y_{t-1}; C) = f(y_{t-1}, g_t, C), \tag{14}$$

where f is a non-linear function, e.g. the tanh function and g_t is the hidden state of the decoder network.

Encoder-decoder architectures are often used for machine translation, where they excel phrase-based translation models (Sutskever et al. (2014)). They found that separate deep LSTM RNNs for encoder and decoder outperformed shallow architectures. When translating sentences a useful trick is to revert the order of words in the input sequence (but not the target sequence). This way, the first words of the input sentence and the first words of the output sentence are closer together. Though the average distance between words is the same, the minimal time lag between words from the input sequence and output sequence is much smaller. Sutskever et al. (2014) found that not only is the translation of the first words more accurate, but more surprisingly this also gives better performance on long sentences. Figure 11 shows an encoder-decoder network for machine translation.

The encoder network is not necessarily a recurrent network. For example in image captioning the encoder network could be a convolutional neural network, which extracts relevant features from the input image. Then the decoder-RNN can focus on these extracted regions and generate image descriptions. See Karpathy and Li (2014) for more details.

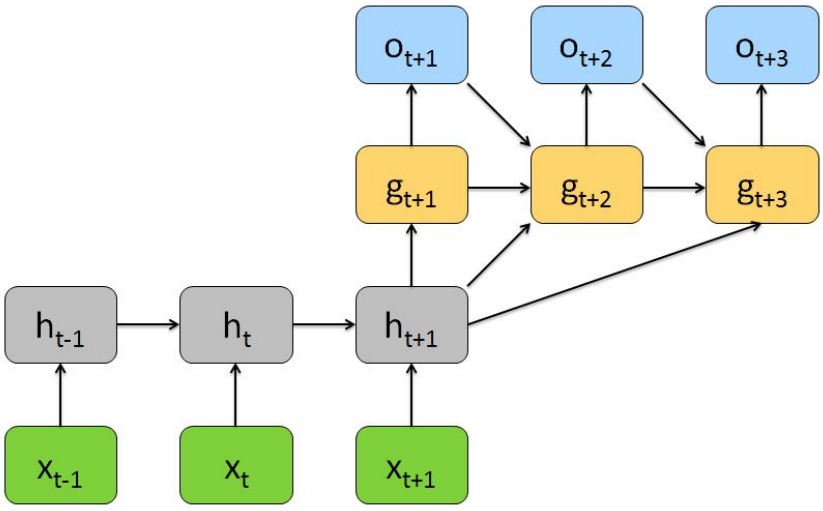


Figure 11: Encoder-decoder allows an RNN to process different length input and output sequences. In the first part of the network information from the input is encoded in the context, here the final hidden state, which is then passed on to every hidden state of the decoder, which produces the target sequence.

5.9 Attention

Bahdanau et al. (2014) 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_{n_x}, which is then passed as an input to the decoder network. With a long input sequence this fixed-sized context vector is unlikely to capture all relevant information about the past. Performance can be greatly improved by using an attention mechanism. It allows the decoder network to focus on different parts of the input sequence by adding connections from all hidden states of the encoder to each hidden state of the decoder.

At each point in time a set of weights is computed, which determine, how to combine the hidden states of the encoder into a context vector c_i, which holds the necessary information to predict the correct output. Each hidden state contains mostly information from recent inputs, in the case of a bidirectional RNN to encode the input sequence, a hidden state contains information from recent preceding and following inputs (Bahdanau et al. (2014)).

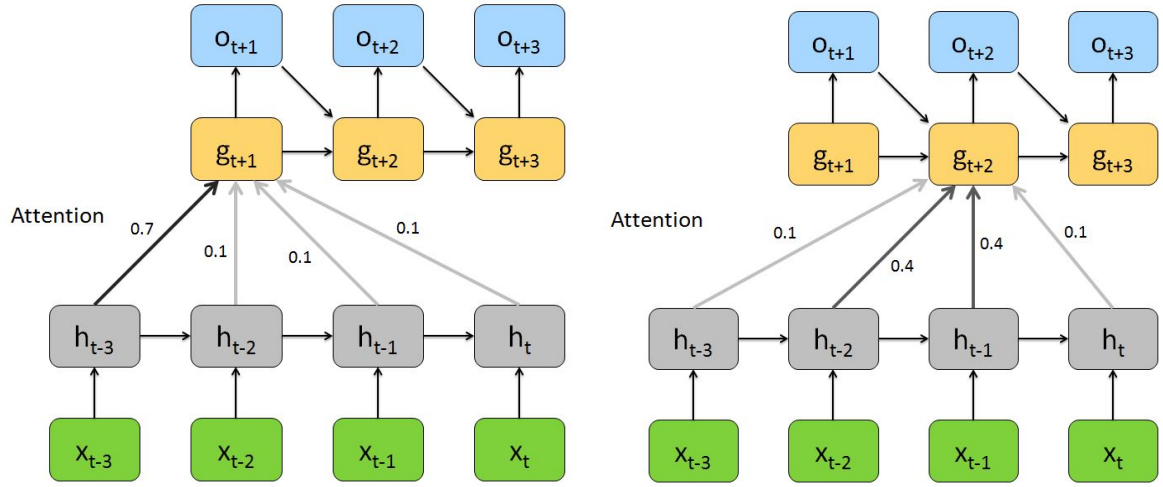
In comparison to the encoder-decoder architecture introduced in Section 5.8, the network now does not have to learn one fixed-sized vector holding all information from the input, but instead uses the set of all hidden states, of which the decoder can choose a subset to extract the relevant information for the current output. For each member in the output sequence there exists now a succinct context vector c_i focusing on different parts of the input:

$$c_i = \sum_{j=1}^{n_x} a_{ij} h_j. \tag{15}$$

The attention weights are computed by a softmax:

$$a_{ij} = \frac{\exp(e_{ij})}{\sum_k^{n_x} e_{ik}} \tag{16}$$

Each a_{ij} can be interpreted as a probability if word i of the target sequence is aligned to word j of the input sequence. The sum over attention weights is then an estimator of an expectation, the expected alignment position, telling us, with which words of the input the current output word is related. The attention weights are usually learned with a feed-forward neural network and depend on ...

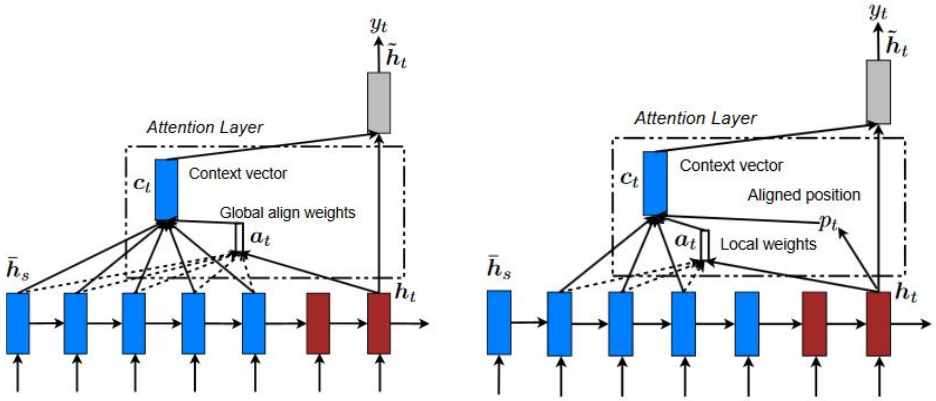


(a) Attention at t=t+1 **(b)** Attention at t=t+2

Figure 12: Attention

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_{n_x} of the input RNN, the attention distribution is therefore a variable-length vector of weights. The context vector c_i is then a weighted average of

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.



(a) Global attention **(b)** Local attention

Figure 13: Attention mechanisms. 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 Application: Google's Neural Machine Translation

For decades machine translation of texts was build upon phrase-based concepts, which translated words and parts of sentences by looking up its translation in a dictionary. With the renaissance of neural networks and lots of promising results in diverse applications like image recognition and other classification tasks it was unsurprising that deep learning could be used to improve performance on translation tasks. Recurrent Neural Networks build the main tool to do so. They have been applied by Bahdanau et al. (2014) and Sutskever et al. (2014) with promising results. So it was only a matter of time till a large AI oriented company like Google would use neural network architectures to improve upon their translation service Google Translate. In September 2016 and November 2016 the Google Research team published two papers, in which a new Recurrent neural network architecture for machine translation was presented, which achieves state-of-the-art results on benchmark data sets. In the following we will have a in-depth look at these architecture.

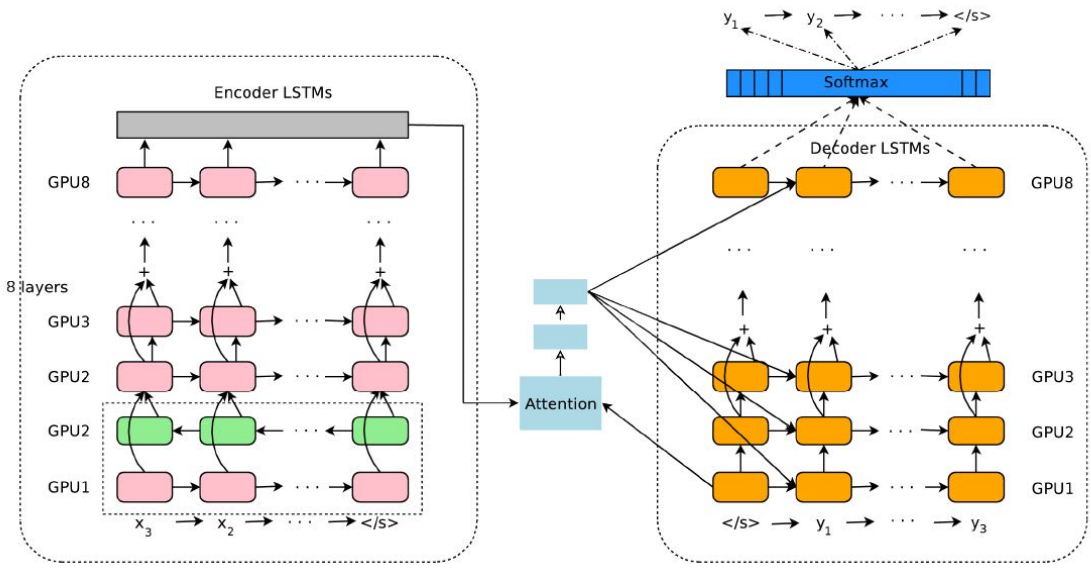


Figure 14: Google's Neural Machine Translation architecture (Wu et al. (2016))

Google's Neural Machine Translation (Wu et al. (2016)) is an application of a Recurrent Neural Network to the task of machine translation achieving state-of-the-art performance.
The architecture consists of an encoder-decoder architecture. Both encoder and decoder are build of several stacked LSTMs, each with 8 layers. In the the first hidden layer of the encoder is a bidirectional RNN, thereafter LSTMs with forward flow of information are stacked. For training this deep architecture residual connections between layers are used. Without residual connections only 4 stacked layers would be trainable, otherwise the problem of vanishing and exploding gradients would be prevent successful training. With residual connections 8 and possibly even more layers can be stacked. An attention mechanism allows the decoder RNN to

focus on different parts of the hidden states of the top layer of the encoder RNN. The attention network is a feed-forward network with one hidden layer computing the attention weights for each time step.
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.
Special tricks had to be developed to deal with three common problems of neural-network-based machine translation. First training and inference speed was noticeably sped up by training in parallel on multiple GPUs. To deal with rare words, words were split into sub-word units (wordpieces) to have a compromise between the efficiency of word-level models and the flexibility of character-level models. To encourage the network to translate all words of the input sentence a coverage penalty was added to the training criterion. At test time a beam search over possible translations was made and the one with highest probability chosen. To prevent the network from favoring short sentences (because for every new word a negative log probability is added), a length normalization term was included in the score function, which is then maximized.

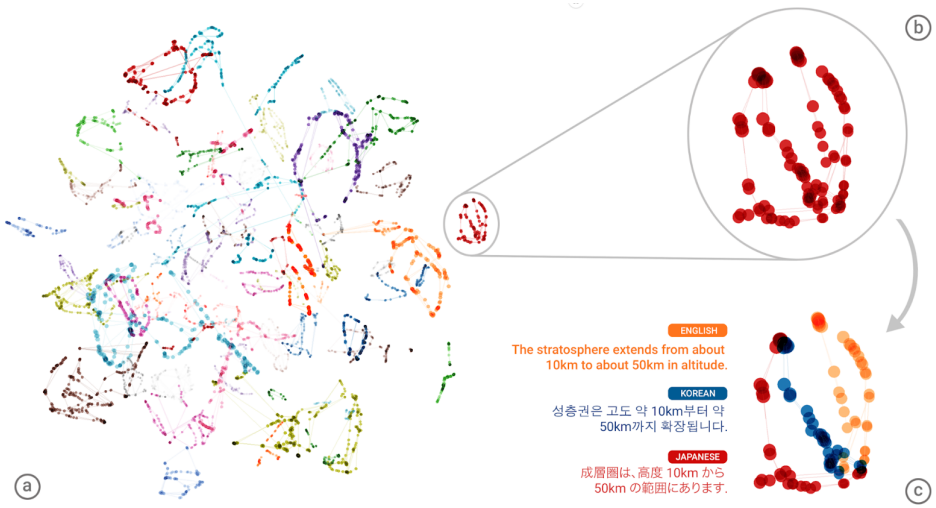


Figure 15: Language embeddings found by Google's Neural Machine Translation (Johnson et al. (2016))

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.

References

Bahdanau, D., Cho, K. and Bengio, Y. (2014). Neural machine translation by jointly learning to align and translate, *CoRR* **abs/1409.0473**.
URL: *http://arxiv.org/abs/1409.0473*

Cho, K., van Merrienboer, B., Gülçehre, Ç., Bougares, F., Schwenk, H. and Bengio, Y. (2014). Learning phrase representations using RNN encoder-decoder for statistical machine translation, *CoRR* **abs/1406.1078**.
URL: *http://arxiv.org/abs/1406.1078*

Goodfellow, I., Bengio, Y. and Courville, A. (2016). *Deep Learning*, MIT Press. `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*

Huang, G., Sun, Y., Liu, Z., Sedra, D. and Weinberger, K. Q. (2016). Deep networks with stochastic depth, *CoRR* **abs/1603.09382**.
URL: *http://arxiv.org/abs/1603.09382*

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*

Jozefowicz, R., Zaremba, W. and Sutskever, I. (2015). An empirical exploration of recurrent network architectures, *Journal of Machine Learning Research* .

Karpathy, A. and Li, F. (2014). Deep visual-semantic alignments for generating image descriptions, *CoRR* **abs/1412.2306**.
URL: *http://arxiv.org/abs/1412.2306*

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*

Socher, R., Perelygin, A., Wu, J., Chuang, J., Manning, C. D., Ng, A. Y. and Potts, C. (2013). Recursive deep models for semantic compositionality over a sentiment treebank, *Proceedings of the 2013 Conference on Empirical Methods in Natural Language Processing*, Association for Computational Linguistics, Stroudsburg, PA, pp. 1631–1642.

Sutskever, I., Vinyals, O. and Le, Q. V. (2014). Sequence to sequence learning with neural networks, *CoRR* **abs/1409.3215**.
URL: *http://arxiv.org/abs/1409.3215*

Veit, A., Wilber, M. J. and Belongie, S. J. (2016). Residual networks are exponential ensembles of relatively shallow networks, *CoRR* **abs/1605.06431**.
URL: *http://arxiv.org/abs/1605.06431*

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