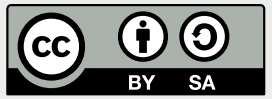
Recurrent Neural Networks

Milan Straka

April 14, 2020

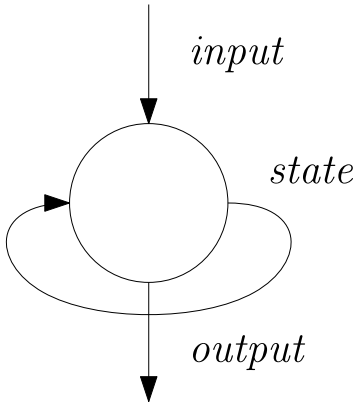
EUROPEAN UNION
European Structural and Investment Fund
Operational Programme Research,
Development and Education

Charles University in Prague
Faculty of Mathematics and Physics
Institute of Formal and Applied Linguistics

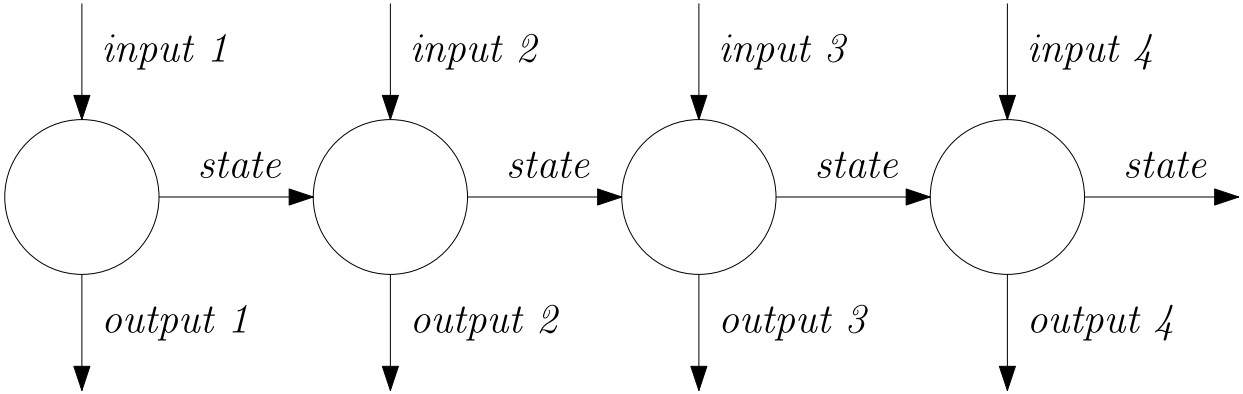


Recurrent Neural Networks

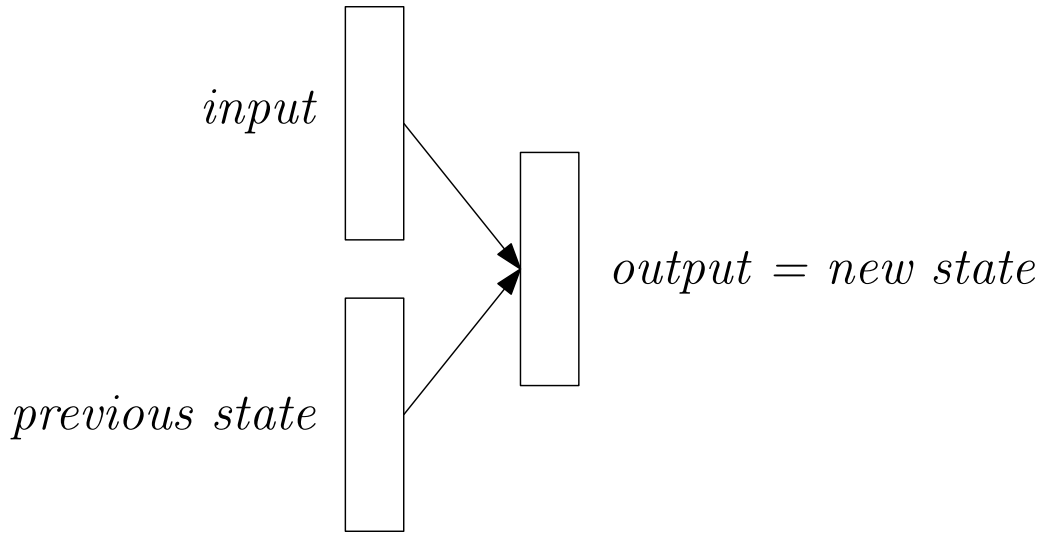
Single RNN cell



Unrolled RNN cells



Basic RNN Cell



Given an input $\boldsymbol{x}^{(t)}$ and previous state $\boldsymbol{s}^{(t-1)}$, the new state is computed as

$$\boldsymbol{s}^{(t)} = f(\boldsymbol{s}^{(t-1)}, \boldsymbol{x}^{(t)}; \boldsymbol{\theta}).$$

One of the simplest possibilities (called `SimpleRNN` in TensorFlow) is

$$\boldsymbol{s}^{(t)} = \tanh(\boldsymbol{U}\boldsymbol{s}^{(t-1)} + \boldsymbol{V}\boldsymbol{x}^{(t)} + \boldsymbol{b}).$$

Basic RNN Cell

Basic RNN cells suffer a lot from vanishing/exploding gradients (*the challenge of long-term dependencies*).

If we simplify the recurrence of states to

$$\boldsymbol{s}^{(t)} = \boldsymbol{U}\boldsymbol{s}^{(t-1)},$$

we get

$$\boldsymbol{s}^{(t)} = \boldsymbol{U}^t\boldsymbol{s}^{(0)}.$$

If U has eigenvalue decomposition of $\boldsymbol{U} = \boldsymbol{Q}\boldsymbol{\Lambda}\boldsymbol{Q}^{-1}$, we get

$$\boldsymbol{s}^{(t)} = \boldsymbol{Q}\boldsymbol{\Lambda}^t\boldsymbol{Q}^{-1}\boldsymbol{s}^{(0)}.$$

The main problem is that the *same* function is iteratively applied many times.

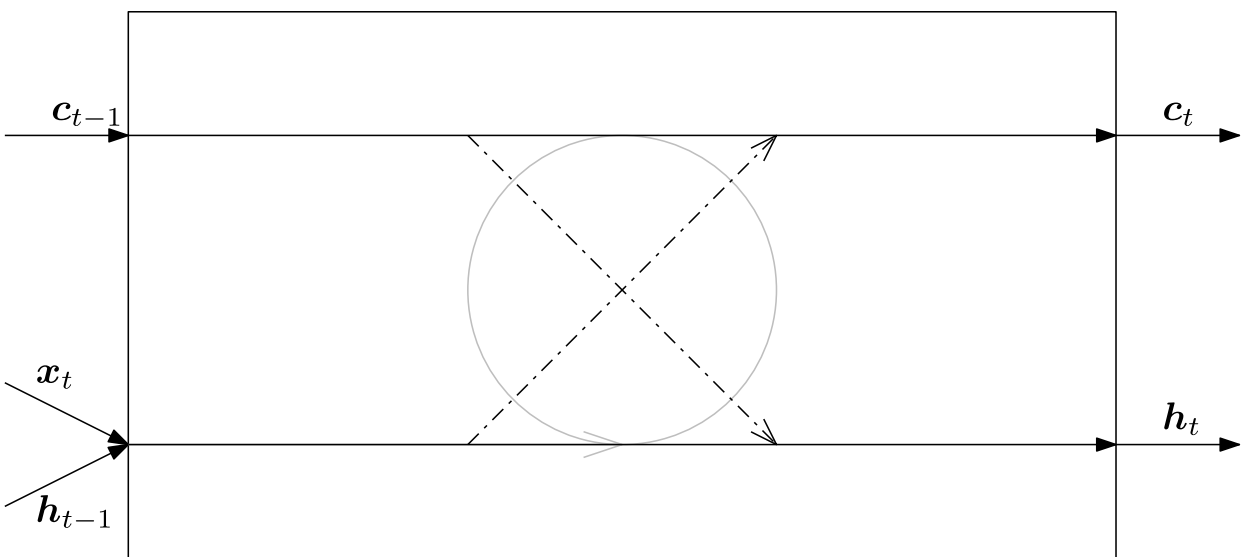
Several more complex RNN cell variants have been proposed, which alleviate this issue to some degree, namely **LSTM** and **GRU**.

Long Short-Term Memory

Hochreiter & Schmidhuber (1997) suggested that to enforce *constant error flow*, we would like

$$f' = \mathbf{1}.$$

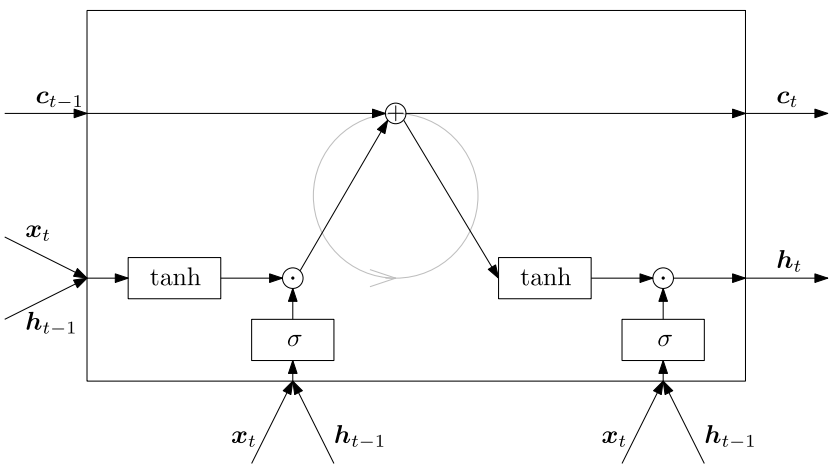
They propose to achieve that by a *constant error carrousel*.



Long Short-Term Memory

They also propose an *input* and *output* gates which control the flow of information into and out of the carrousel (*memory cell* $\boldsymbol{c}_t$).

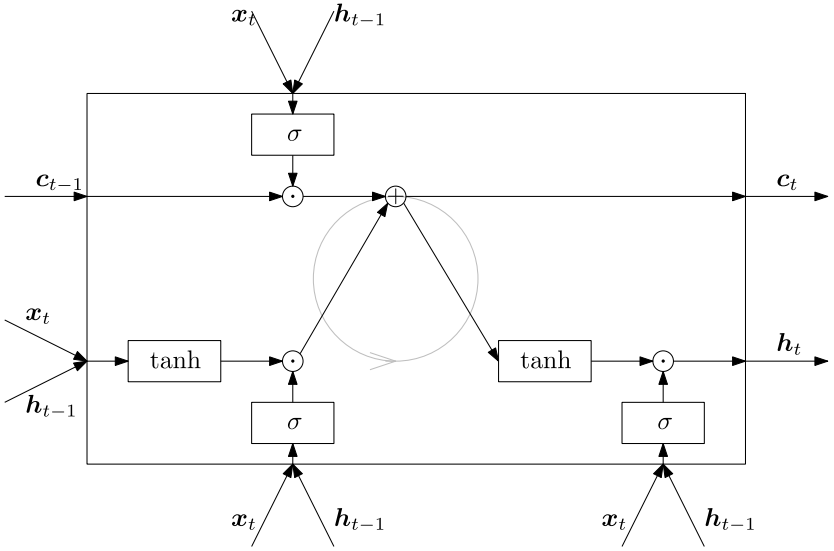
$$\boldsymbol{i}_t \leftarrow \sigma(\boldsymbol{W}^i \boldsymbol{x}_t + \boldsymbol{V}^i \boldsymbol{h}_{t-1} + \boldsymbol{b}^i)$$
$$\boldsymbol{o}_t \leftarrow \sigma(\boldsymbol{W}^o \boldsymbol{x}_t + \boldsymbol{V}^o \boldsymbol{h}_{t-1} + \boldsymbol{b}^o)$$
$$\boldsymbol{c}_t \leftarrow \boldsymbol{c}_{t-1} + \boldsymbol{i}_t \cdot \tanh(\boldsymbol{W}^y \boldsymbol{x}_t + \boldsymbol{V}^y \boldsymbol{h}_{t-1} + \boldsymbol{b}^y)$$
$$\boldsymbol{h}_t \leftarrow \boldsymbol{o}_t \cdot \tanh(\boldsymbol{c}_t)$$



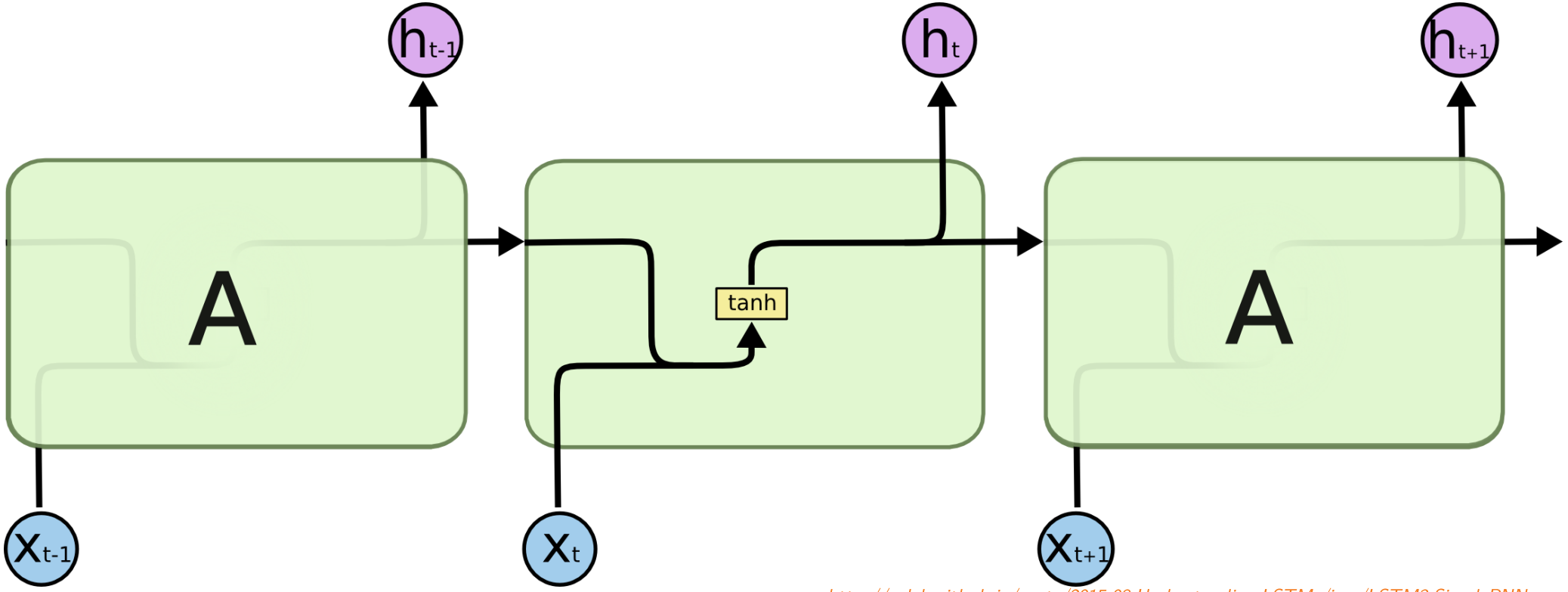
Long Short-Term Memory

Later in Gers, Schmidhuber & Cummins (1999) a possibility to *forget* information from memory cell $\boldsymbol{c}_t$ was added.

$$\begin{aligned}
\boldsymbol{i}_t &\leftarrow \sigma(\boldsymbol{W}^i \boldsymbol{x}_t + \boldsymbol{V}^i \boldsymbol{h}_{t-1} + \boldsymbol{b}^i) \\
\boldsymbol{f}_t &\leftarrow \sigma(\boldsymbol{W}^f \boldsymbol{x}_t + \boldsymbol{V}^f \boldsymbol{h}_{t-1} + \boldsymbol{b}^f) \\
\boldsymbol{o}_t &\leftarrow \sigma(\boldsymbol{W}^o \boldsymbol{x}_t + \boldsymbol{V}^o \boldsymbol{h}_{t-1} + \boldsymbol{b}^o) \\
\boldsymbol{c}_t &\leftarrow \boldsymbol{f}_t \cdot \boldsymbol{c}_{t-1} + \boldsymbol{i}_t \cdot \tanh(\boldsymbol{W}^y \boldsymbol{x}_t + \boldsymbol{V}^y \boldsymbol{h}_{t-1} + \boldsymbol{b}^y) \\
\boldsymbol{h}_t &\leftarrow \boldsymbol{o}_t \cdot \tanh(\boldsymbol{c}_t)
\end{aligned}$$

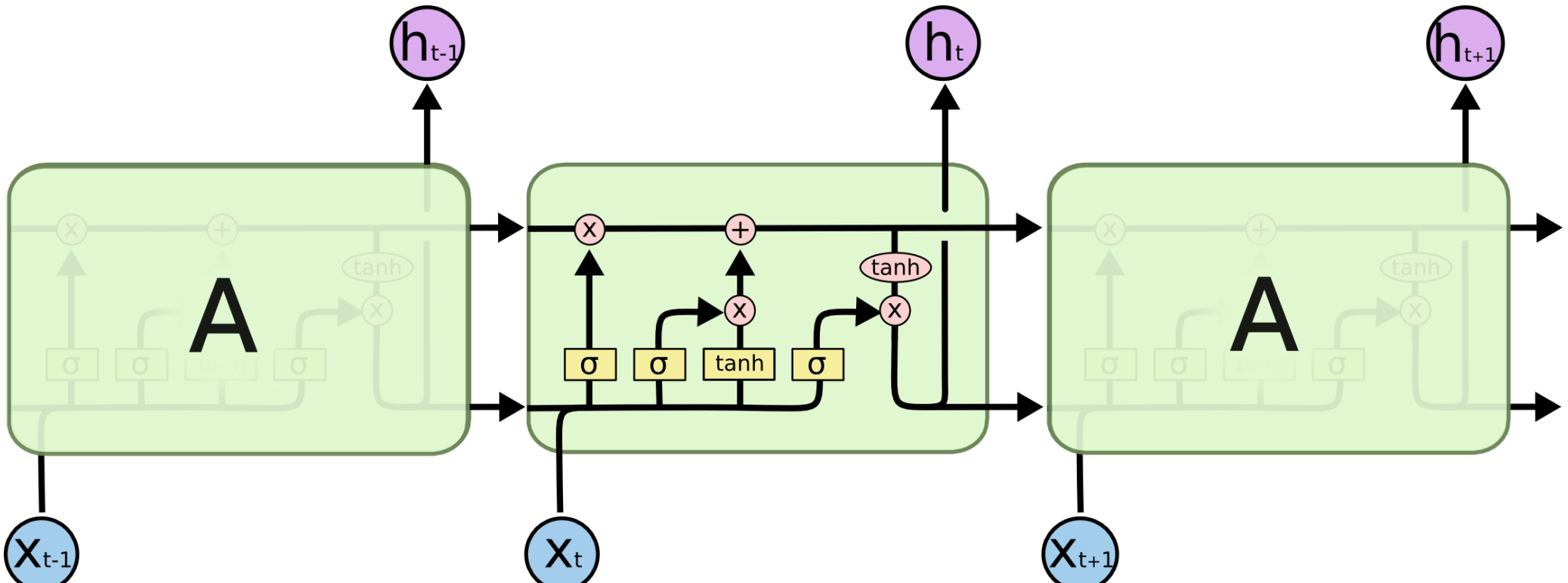


Long Short-Term Memory



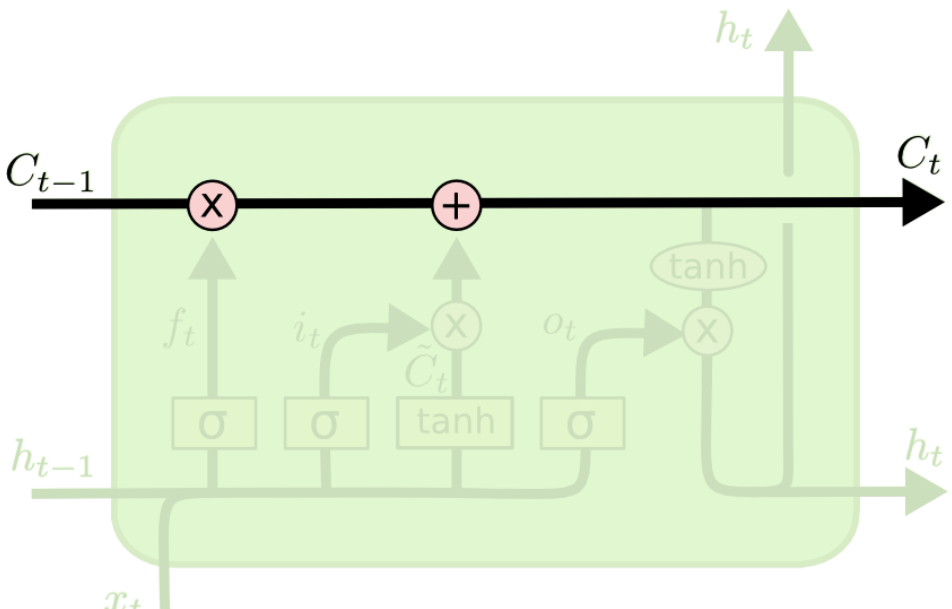
http://colah.github.io/posts/2015-08-Understanding-LSTMs/img/LSTM3-SimpleRNN.png

Long Short-Term Memory



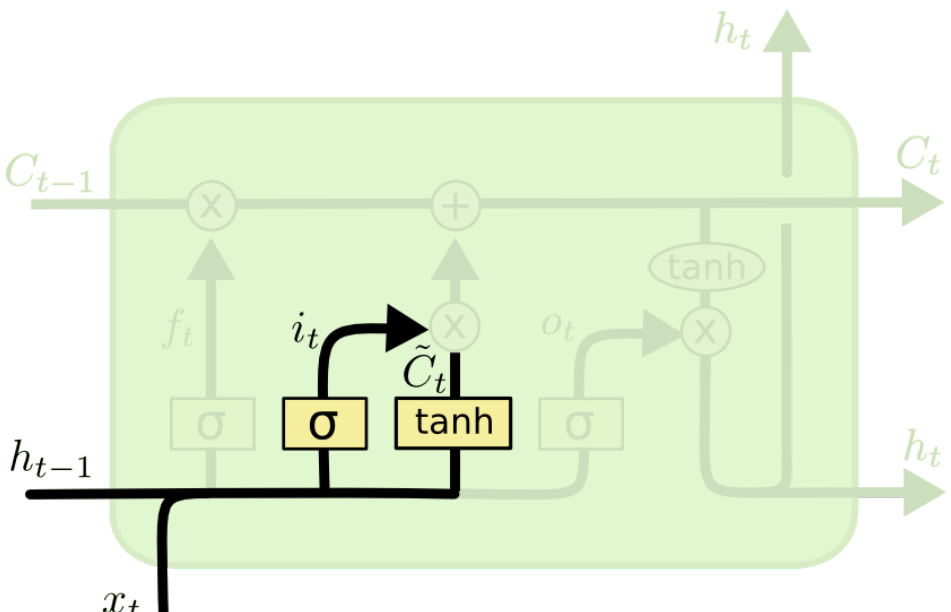
http://colah.github.io/posts/2015-08-Understanding-LSTMs/img/LSTM3-chain.png

Long Short-Term Memory



http://colah.github.io/posts/2015-08-Understanding-LSTMs/img/LSTM3-C-line.png

Long Short-Term Memory

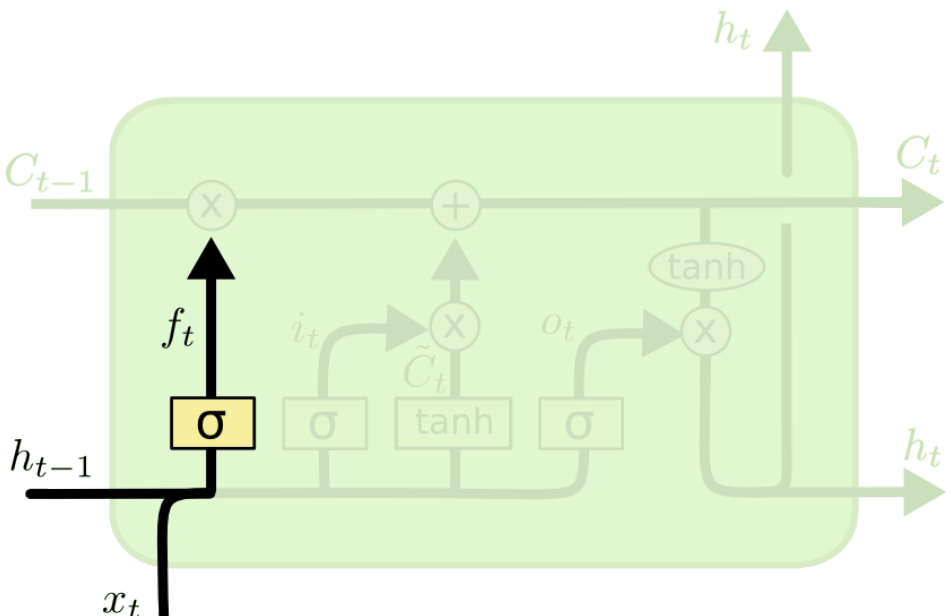


$$i_t = \sigma\left(W_i \cdot [h_{t-1}, x_t] \; + \; b_i\right)$$

$$\tilde{C}_t = \tanh(W_C \cdot [h_{t-1}, x_t] \; + \; b_C)$$

http://colah.github.io/posts/2015-08-Understanding-LSTMs/img/LSTM3-focus-i.png

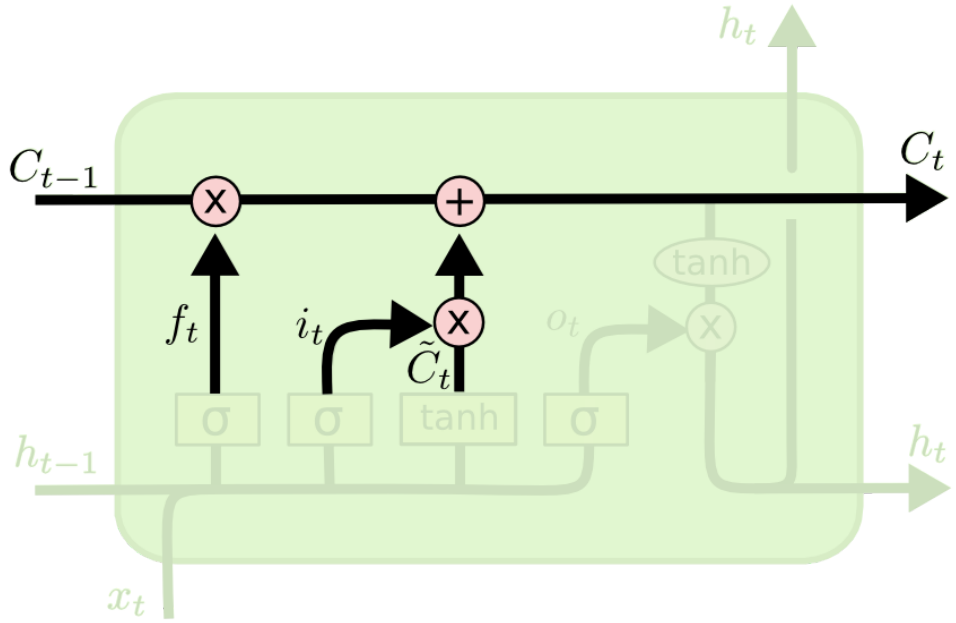
Long Short-Term Memory



$$f_t = \sigma\left(W_f \cdot [h_{t-1}, x_t] \; + \; b_f\right)$$

http://colah.github.io/posts/2015-08-Understanding-LSTMs/img/LSTM3-focus-f.png

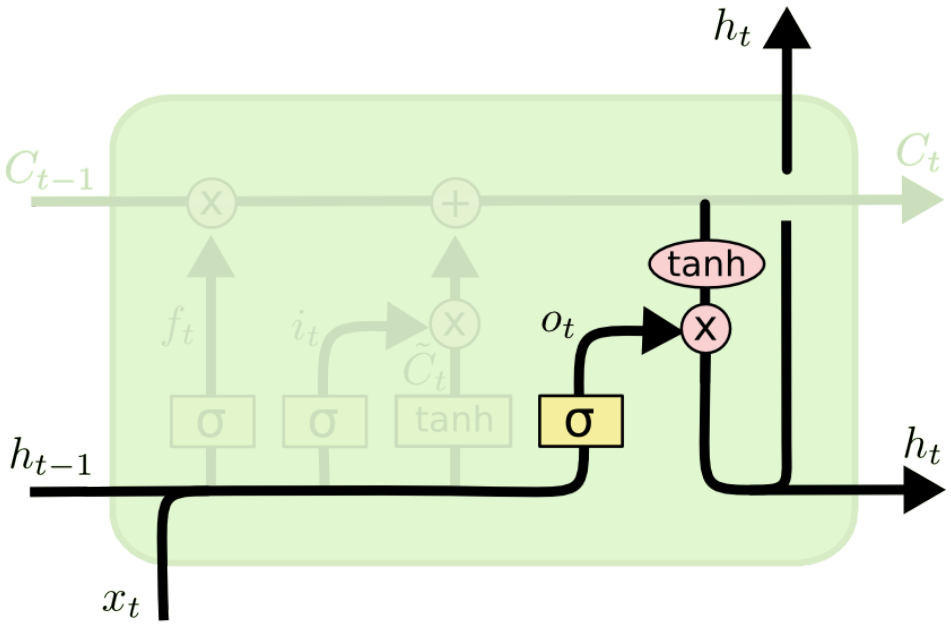
Long Short-Term Memory



$$C_t = f_t * C_{t-1} + i_t * \tilde{C}_t$$

http://colah.github.io/posts/2015-08-Understanding-LSTMs/img/LSTM3-focus-C.png

Long Short-Term Memory



$$o_t = \sigma\left(W_o\left[h_{t-1}, x_t\right] + b_o\right)$$

$$h_t = o_t * \tanh\left(C_t\right)$$

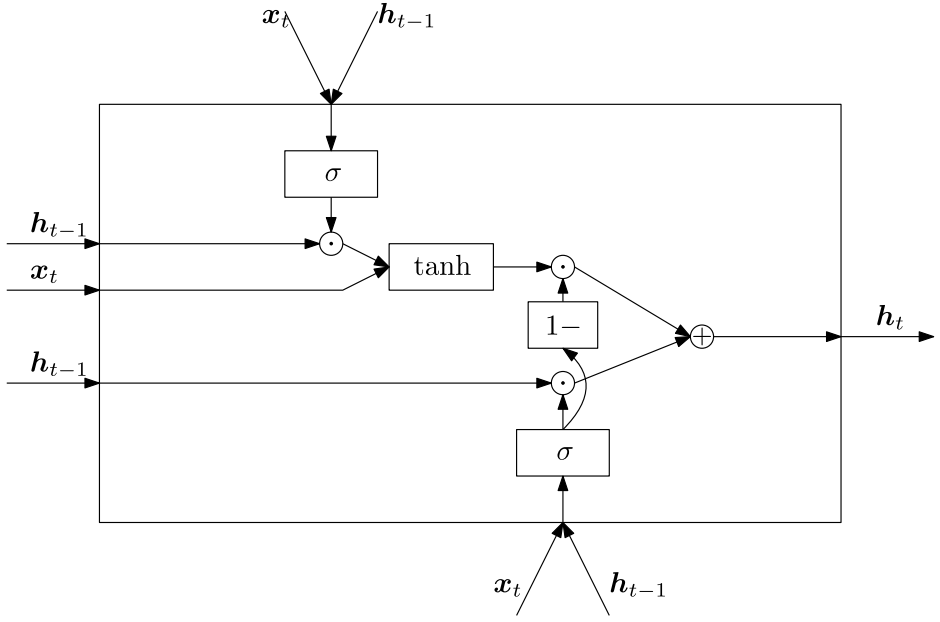
http://colah.github.io/posts/2015-08-Understanding-LSTMs/img/LSTM3-focus-o.png

Gated Recurrent Unit

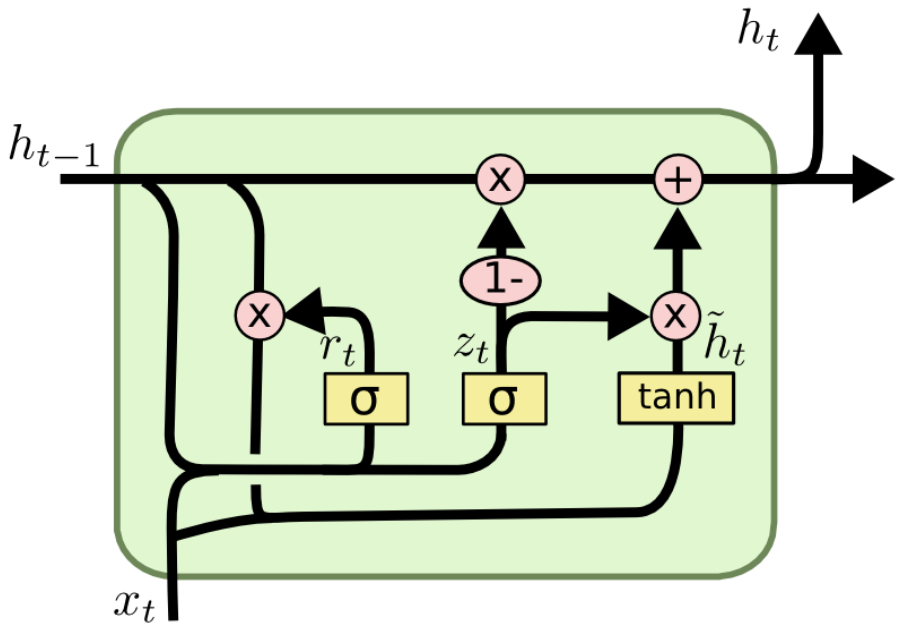
Gated recurrent unit (GRU) was proposed by Cho et al. (2014) as a simplification of LSTM. The main differences are

- no memory cell
- forgetting and updating tied together

$$\boldsymbol{r}_t \leftarrow \sigma(\boldsymbol{W}^r \boldsymbol{x}_t + \boldsymbol{V}^r \boldsymbol{h}_{t-1} + \boldsymbol{b}^r)$$
$$\boldsymbol{u}_t \leftarrow \sigma(\boldsymbol{W}^u \boldsymbol{x}_t + \boldsymbol{V}^u \boldsymbol{h}_{t-1} + \boldsymbol{b}^u)$$
$$\hat{\boldsymbol{h}}_t \leftarrow \tanh(\boldsymbol{W}^h \boldsymbol{x}_t + \boldsymbol{V}^h (\boldsymbol{r}_t \cdot \boldsymbol{h}_{t-1}) + \boldsymbol{b}^h)$$
$$\boldsymbol{h}_t \leftarrow \boldsymbol{u}_t \cdot \boldsymbol{h}_{t-1} + (1 - \boldsymbol{u}_t) \cdot \hat{\boldsymbol{h}}_t$$



Gated Recurrent Unit



$$z_t = \sigma\left(W_z \cdot [h_{t-1}, x_t]\right)$$

$$r_t = \sigma\left(W_r \cdot [h_{t-1}, x_t]\right)$$

$$\tilde{h}_t = \tanh\left(W \cdot [r_t * h_{t-1}, x_t]\right)$$

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

http://colah.github.io/posts/2015-08-Understanding-LSTMs/img/LSTM3-var-GRU.png

Highway Networks

Highway Networks

For input $\boldsymbol{x}$, fully connected layer computes

$$\boldsymbol{y} \leftarrow H(\boldsymbol{x}, \boldsymbol{W}_H).$$

Highway networks add residual connection with gating:

$$\boldsymbol{y} \leftarrow H(\boldsymbol{x}, \boldsymbol{W}_H) \cdot T(\boldsymbol{x}, \boldsymbol{W}_T) + \boldsymbol{x} \cdot (1 - T(\boldsymbol{x}, \boldsymbol{W}_T)).$$

Usually, the gating is defined as

$$T(\boldsymbol{x}, \boldsymbol{W}_T) \leftarrow \sigma(\boldsymbol{W}_T \boldsymbol{x} + \boldsymbol{b}_T).$$

Note that the resulting update is very similar to a GRU cell with $\boldsymbol{h}_t$ removed; for a fully connected layer $H(\boldsymbol{x}, \boldsymbol{W}_H) = \tanh(\boldsymbol{W}_H \boldsymbol{x} + \boldsymbol{b}_H)$ it is exactly it.

Highway Networks

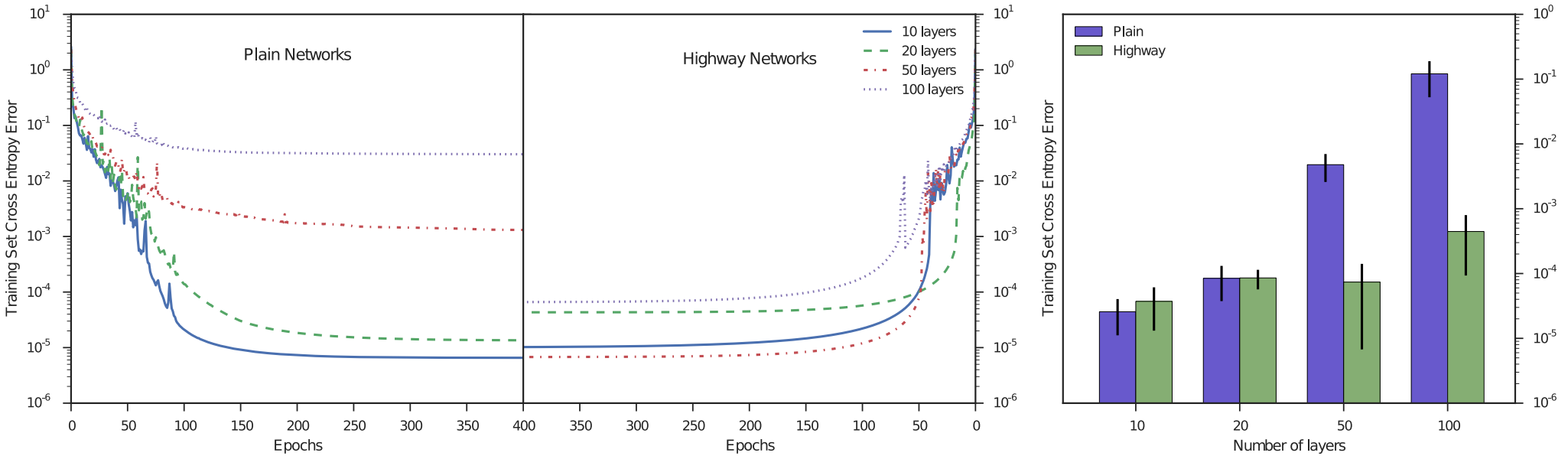


Figure 1: Comparison of optimization of plain networks and highway networks of various depths. *Left:* The training curves for the best hyperparameter settings obtained for each network depth. *Right:* Mean performance of top 10 (out of 100) hyperparameter settings. Plain networks become much harder to optimize with increasing depth, while highway networks with up to 100 layers can still be optimized well. Best viewed on screen (larger version included in Supplementary Material).

Figure 1 of paper "Training Very Deep Networks", https://arxiv.org/abs/1507.06228.

Highway Networks

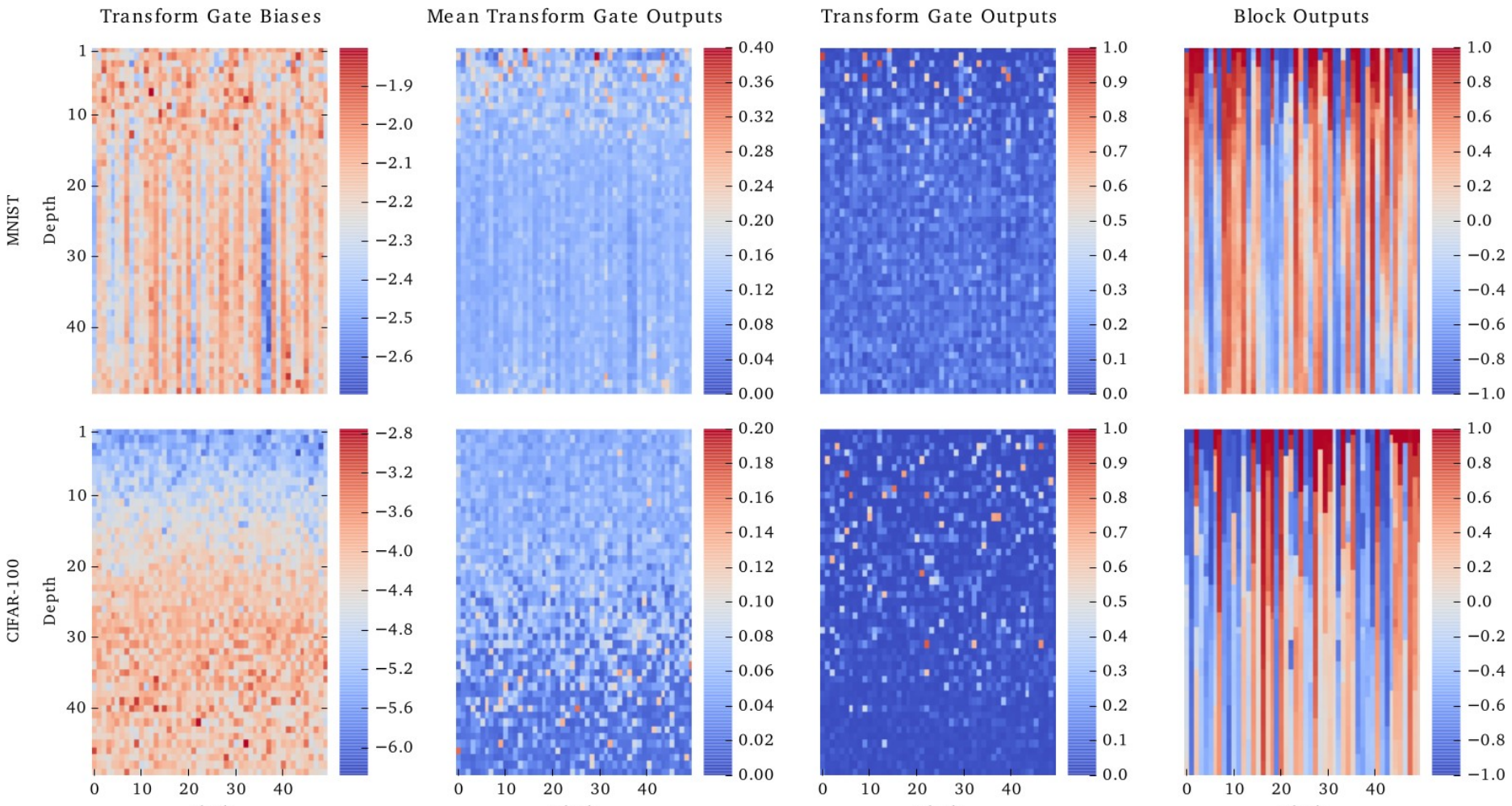


Figure 2 of paper "Training Very Deep Networks", https://arxiv.org/abs/1507.06228.

Highway Networks

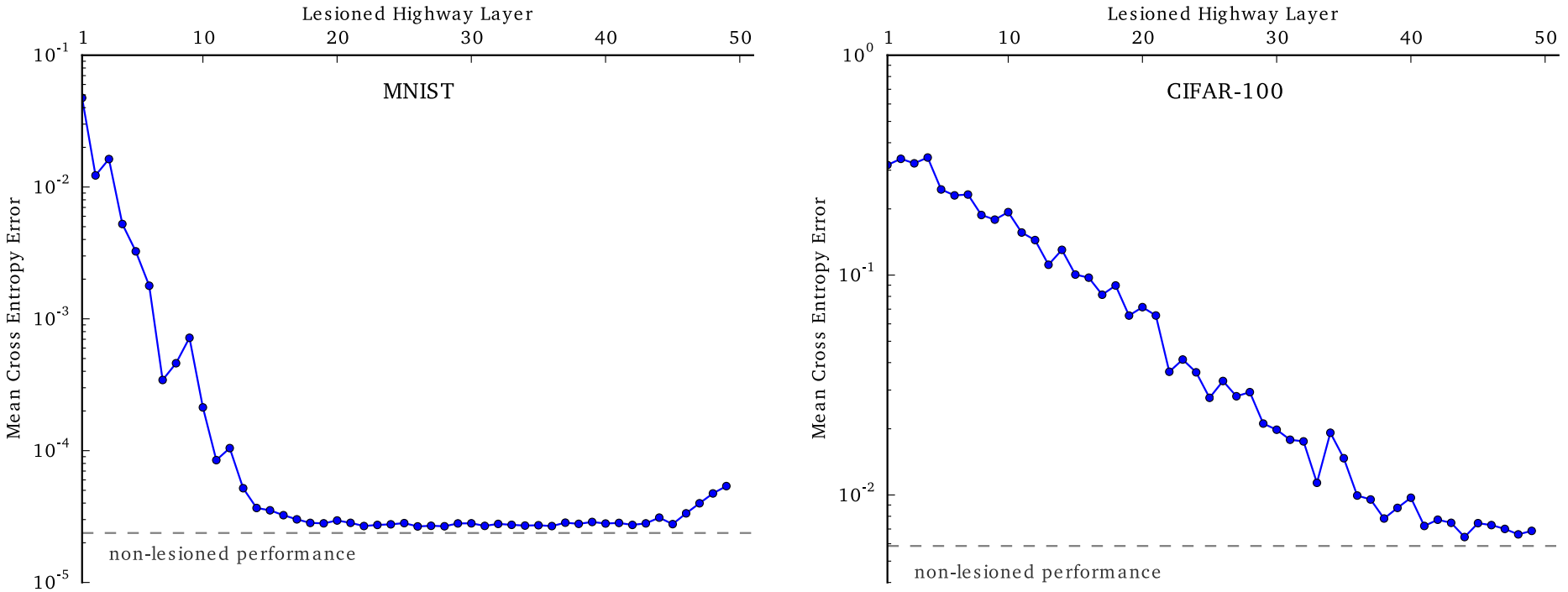


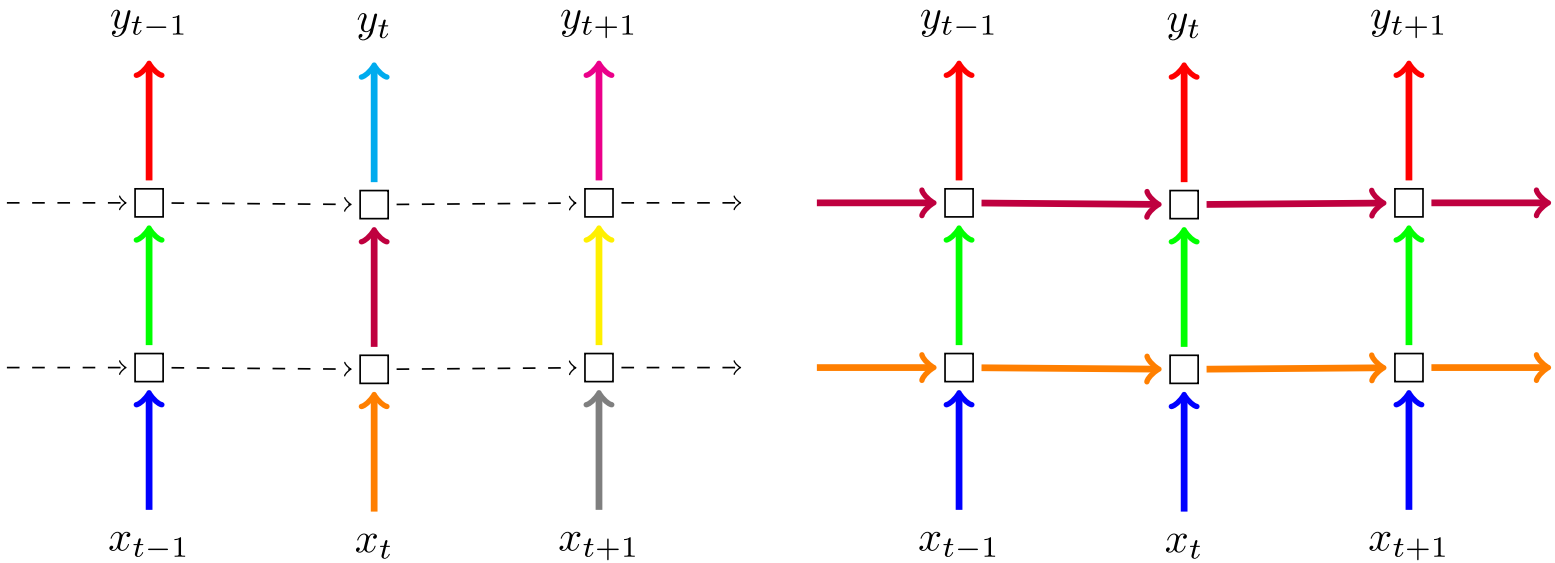
Figure 4: Lesioned training set performance (y-axis) of the best 50-layer highway networks on MNIST (left) and CIFAR-100 (right), as a function of the lesioned layer (x-axis). Evaluated on the full training set while forcefully closing all the transform gates of a single layer at a time. The non-lesioned performance is indicated as a dashed line at the bottom.

Figure 4 of paper "Training Very Deep Networks", https://arxiv.org/abs/1507.06228.

Dropout

- Using dropout on hidden states interferes with long-term dependencies.
- However, using dropout on the inputs and outputs works well and is used frequently.
  - In case residual connections are present, the output dropout needs to be applied before adding the residual connection.
- Several techniques were designed to allow using dropout on hidden states.
  - Variational Dropout
  - Recurrent Dropout
  - Zoneout

Variational Dropout



(a) Naive dropout RNN (b) Variational RNN

Figure 1 of paper "A Theoretically Grounded Application of Dropout in Recurrent Neural Networks", https://arxiv.org/abs/1512.05287.pdf

To implement variational dropout on inputs in TensorFlow, use `noise_shape` of `tf.keras.layers.Dropout` to force the same mask across time-steps. The variational dropout on the hidden states can be implemented using `recurrent_dropout` argument of `tf.keras.layers.{LSTM,GRU,SimpleRNN}{,Cell}`.

Recurrent Dropout

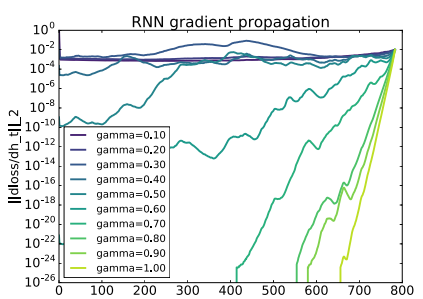
Dropout only candidate states (i.e., values added to the memory cell in LSTM and previous state in GRU).

Zoneout

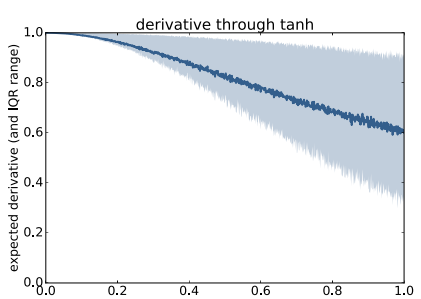
Randomly preserve hidden activations instead of dropping them.

Batch Normalization

Very fragile and sensitive to proper initialization – there were papers with negative results (*Dario Amodei et al, 2015: Deep Speech 2* or *Cesar Laurent et al, 2016: Batch Normalized Recurrent Neural Networks*) until people managed to make it work (*Tim Cooijmans et al, 2016: Recurrent Batch Normalization*; specifically, initializing $\gamma = 0.1$ did the trick).



(a) We visualize the gradient flow through a batch-normalized tanh RNN as a function of γ. High variance causes vanishing gradient.



(b) We show the empirical expected derivative and interquartile range of tanh nonlinearity as a function of input variance. High variance causes saturation, which decreases the expected derivative.

Figure 1 of paper "Recurrent Batch Normalization", https://arxiv.org/abs/1603.09025.

Layer Normalization

Much more stable than batch normalization.

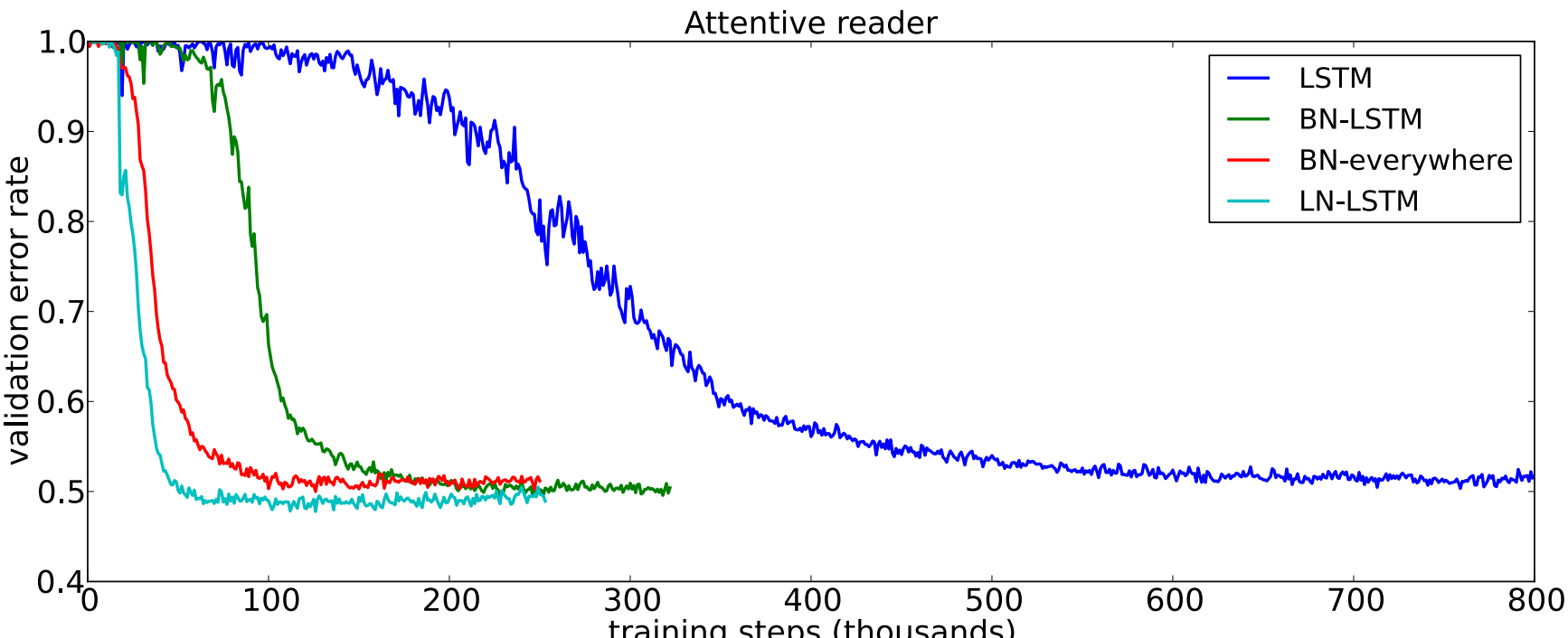
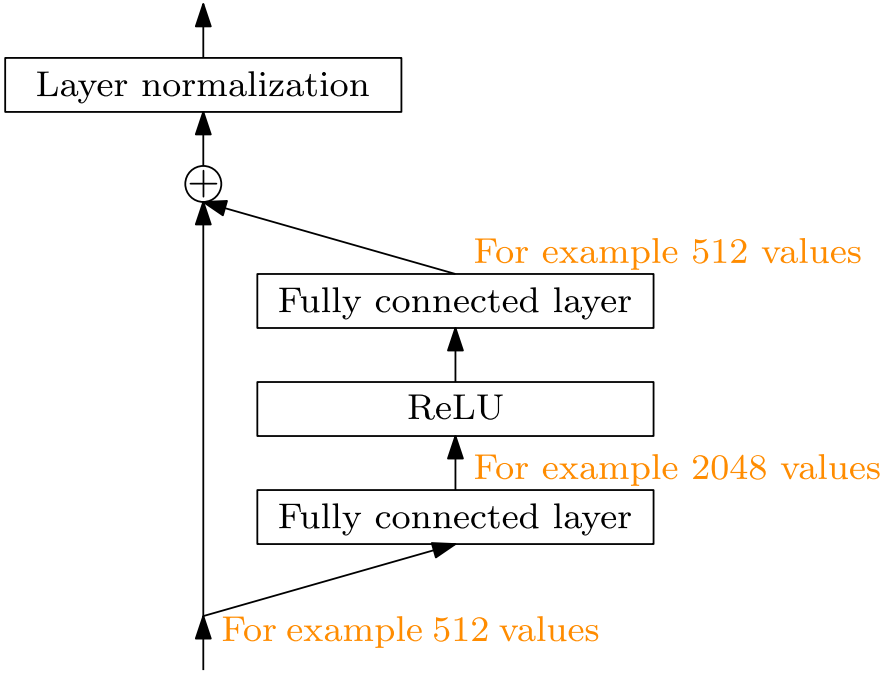


Figure 2: Validation curves for the attentive reader model. BN results are taken from [Cooijmans et al., 2016].

Figure 2 of paper "Layer Normalization", https://arxiv.org/abs/1607.06450.

Layer Normalization

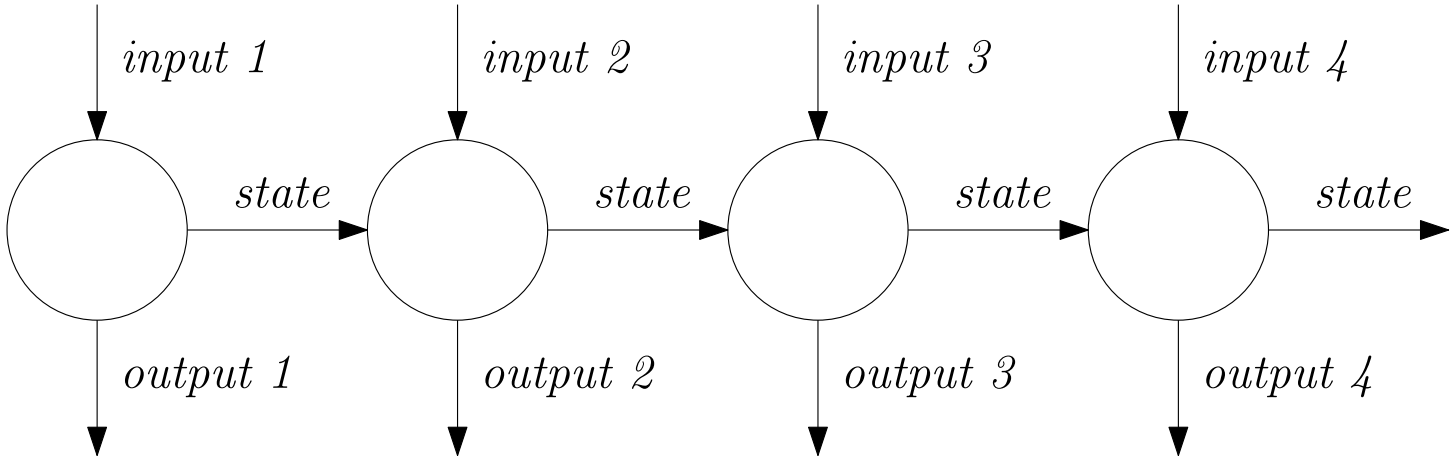
In an important recent architecture (namely Transformer), many fully connected layers are used, with a residual connection and a layer normalization.



This could be considered an alternative to highway networks, i.e., a suitable residual connection for fully connected layers. Note the architecture can be considered as a variant of a mobile inverted bottleneck 1×1 convolution block.

Sequence Element Representation

Create output for individual elements, for example for classification of the individual elements.

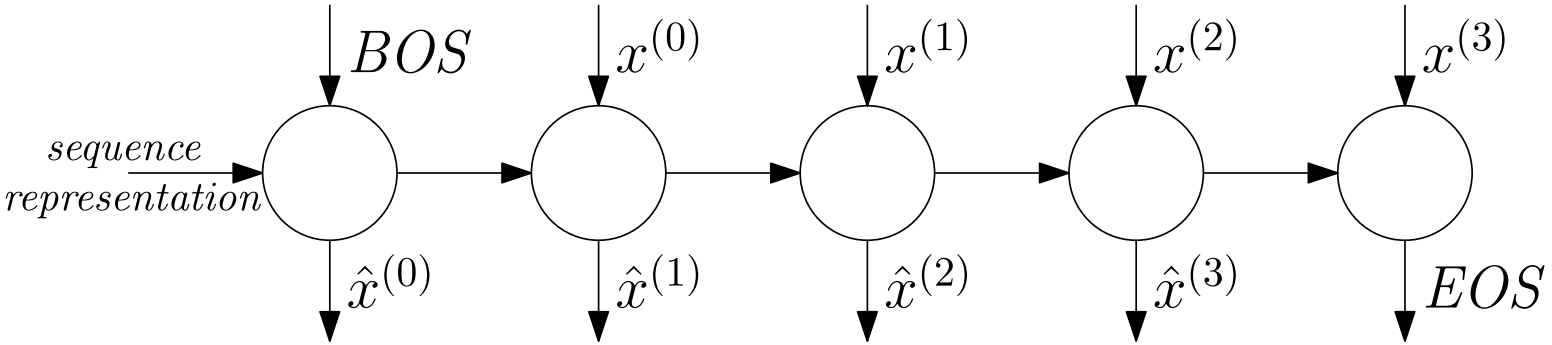


Sequence Representation

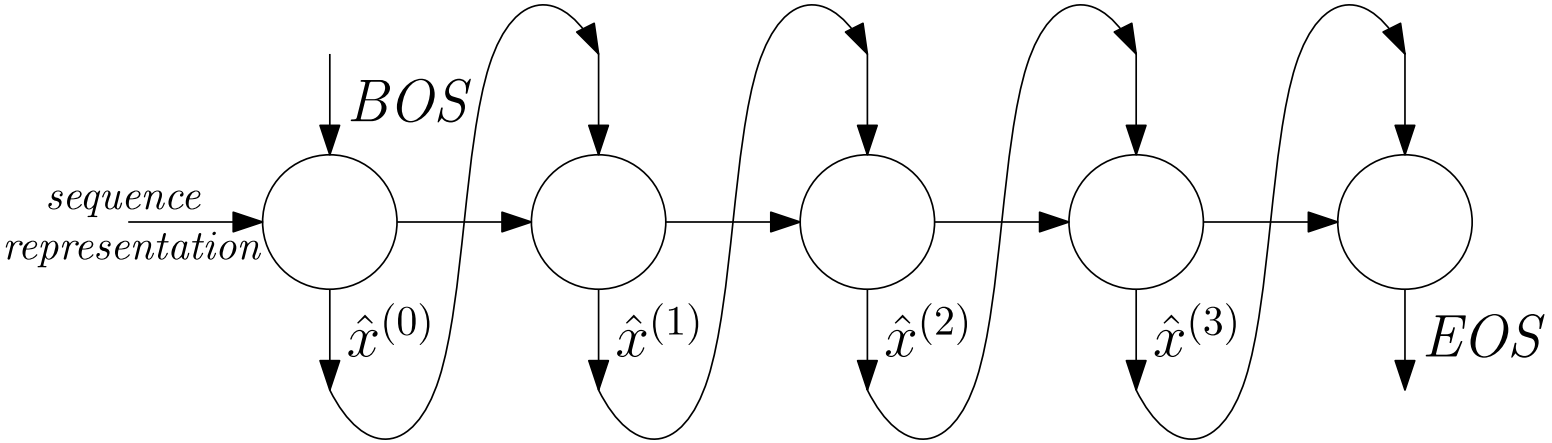
Generate a single output for the whole sequence (either the last output of the last state).

Sequence Prediction

During training, predict next sequence element.

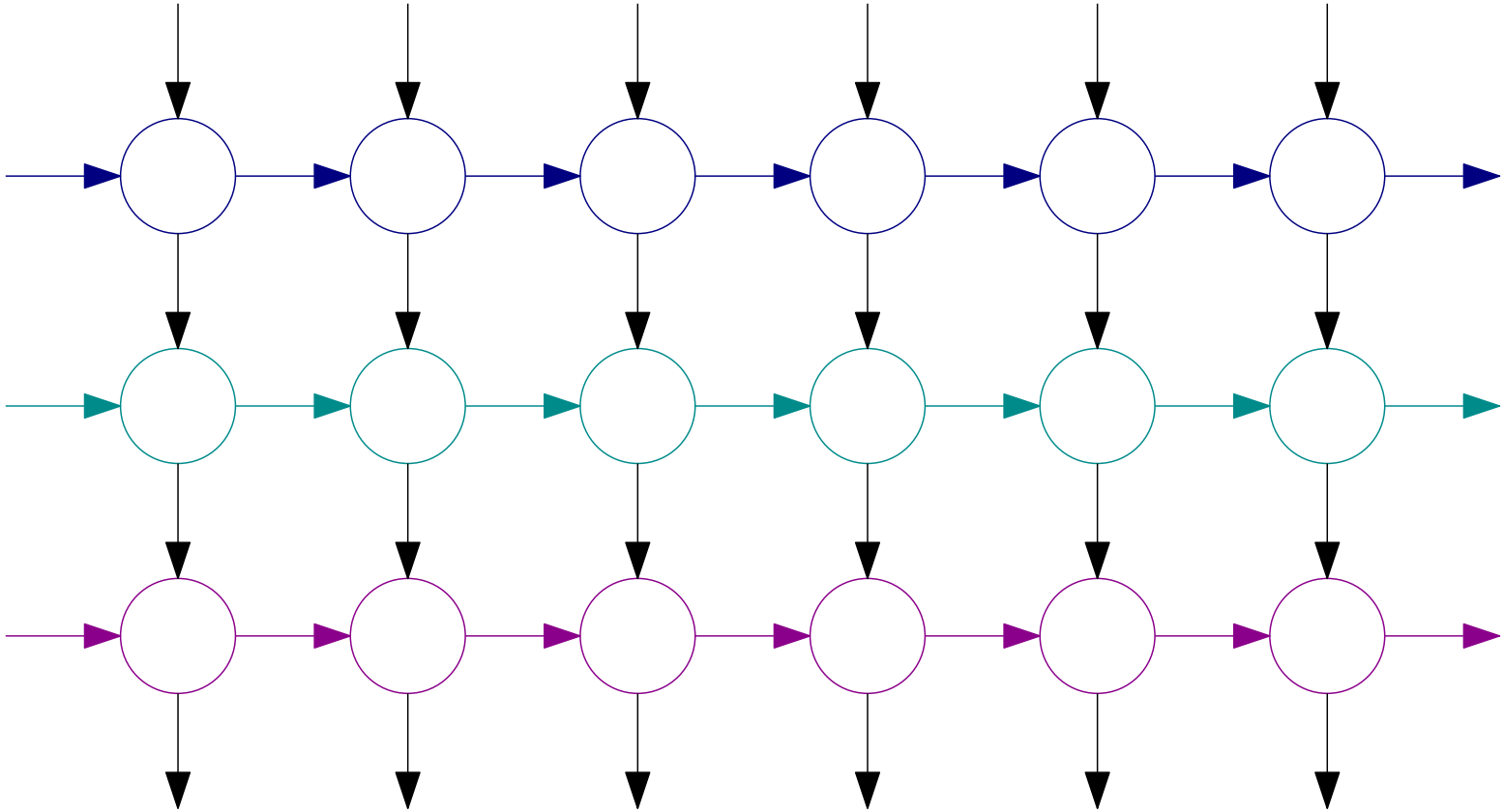


During inference, use predicted elements as further inputs.



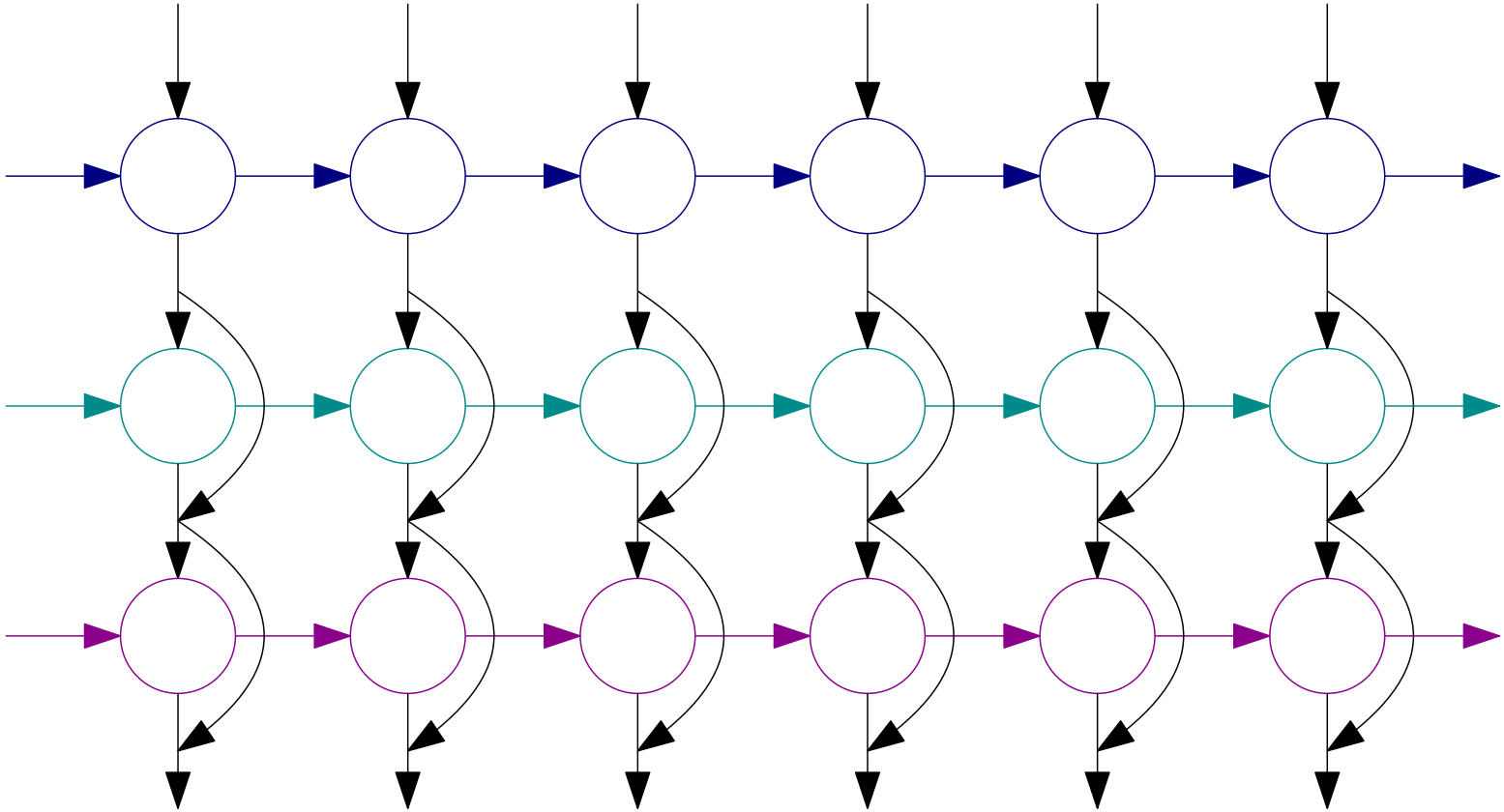
Multilayer RNNs

We might stack several layers of recurrent neural networks. Usually using two or three layers gives better results than just one.



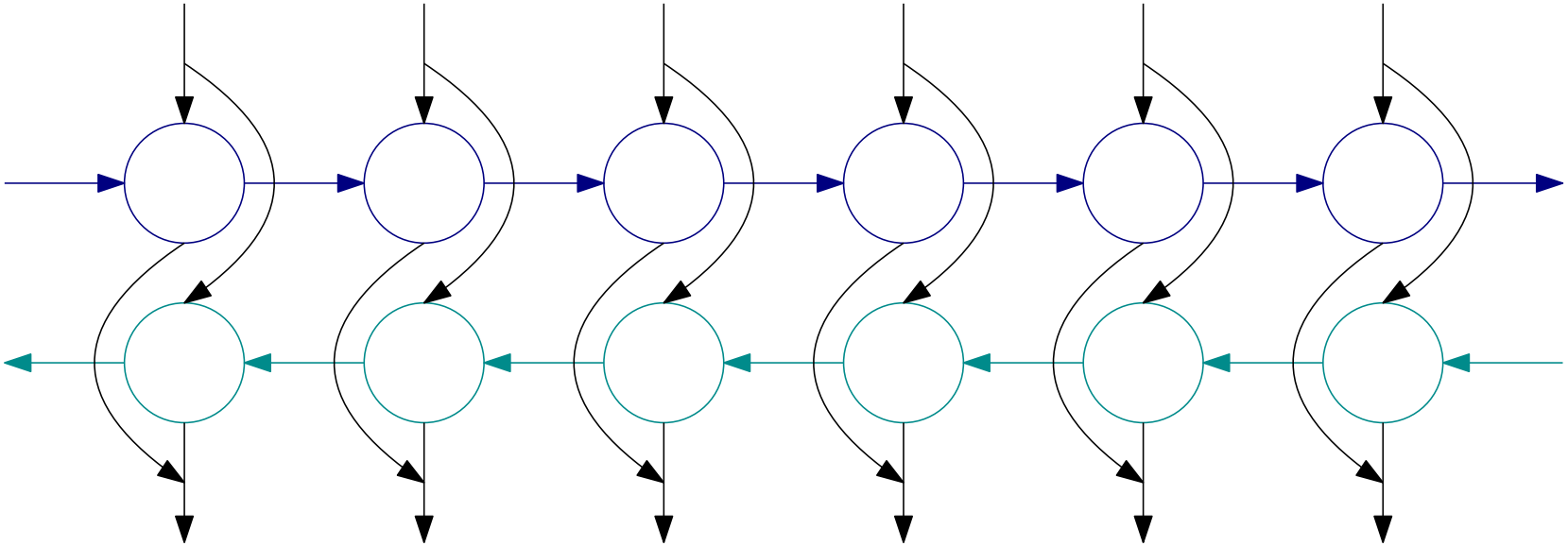
Multilayer RNNs

In case of multiple layers, residual connections usually improve results. Because dimensionality has to be the same, they are usually applied from the second layer.



Bidirectional RNN

To consider both the left and right contexts, a *bidirectional* RNN can be used, which consists of parallel application of a *forward* RNN and a *backward* RNN.



The outputs of both directions can be either *added* or *concatenated*. Even if adding them does not seem very intuitive, it does not increase dimensionality and therefore allows residual connections to be used in case of multilayer bidirectional RNN.

We might represent *words* using one-hot encoding, considering all words to be independent of each other.

However, words are not independent – some are more similar than others.

Ideally, we would like some kind of similarity in the space of the word representations.

Distributed Representation

The idea behind distributed representation is that objects can be represented using a set of common underlying factors.

We therefore represent words as fixed-size *embeddings* into $\mathbb{R}^d$ space, with the vector elements playing role of the common underlying factors.

These embeddings are initialized randomly and trained together with the rest of the network.

Word Embeddings

The word embedding layer is in fact just a fully connected layer on top of one-hot encoding. However, it is not implemented in that way.

Instead, a so-called *embedding* layer is used, which is much more efficient. When a matrix is multiplied by an one-hot encoded vector (all but one zeros and exactly one 1), the row corresponding to that 1 is selected, so the embedding layer can be implemented only as a simple lookup.

In TensorFlow, the embedding layer is available as

```
tf.keras.layers.Embedding(input_dim, output_dim)
```

Word Embeddings

Even if the embedding layer is just a fully connected layer on top of one-hot encoding, it is important that this layer is *shared* across the whole network.