Recurrent Neural Networks II

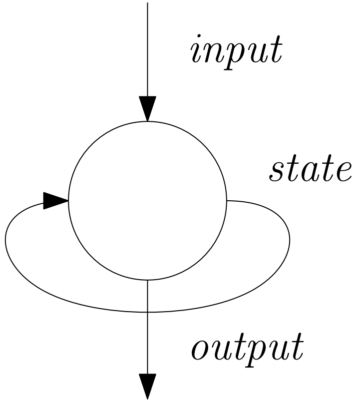
Milan Straka

April 15, 2019

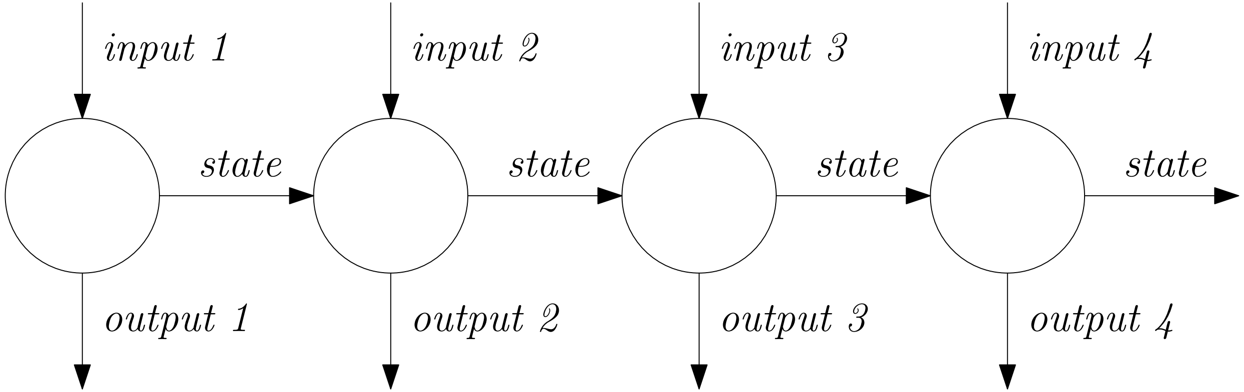
EUROPEAN UNION
European Structural and Investment Fund
Operational Programme Research, Development and Education

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

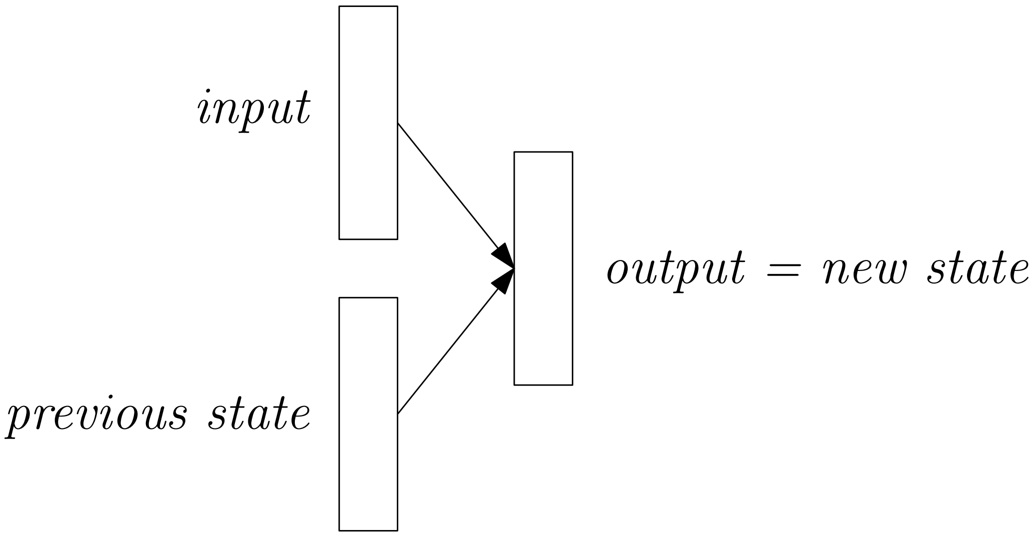
Single RNN cell



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 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

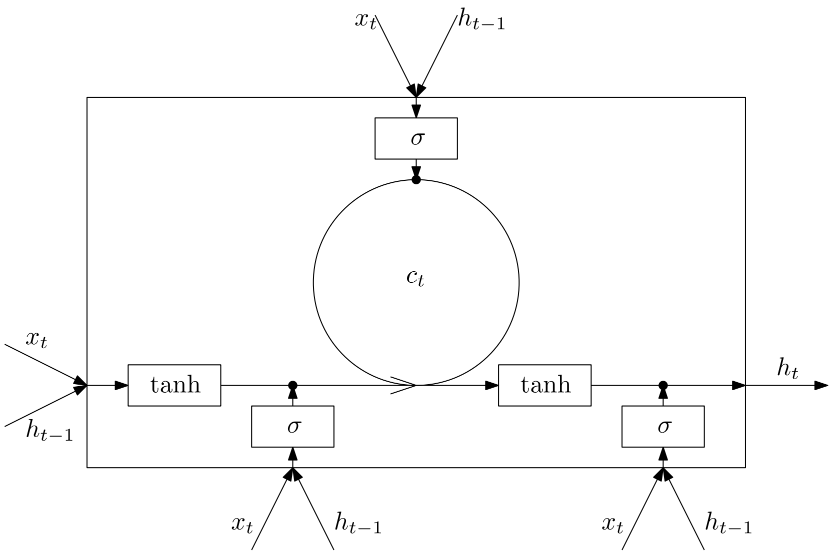
$$\boldsymbol{s}^{(t)} = \boldsymbol{Q}\boldsymbol{\Lambda}^t\boldsymbol{Q}^{-1}\boldsymbol{s}^{(0)}.$$

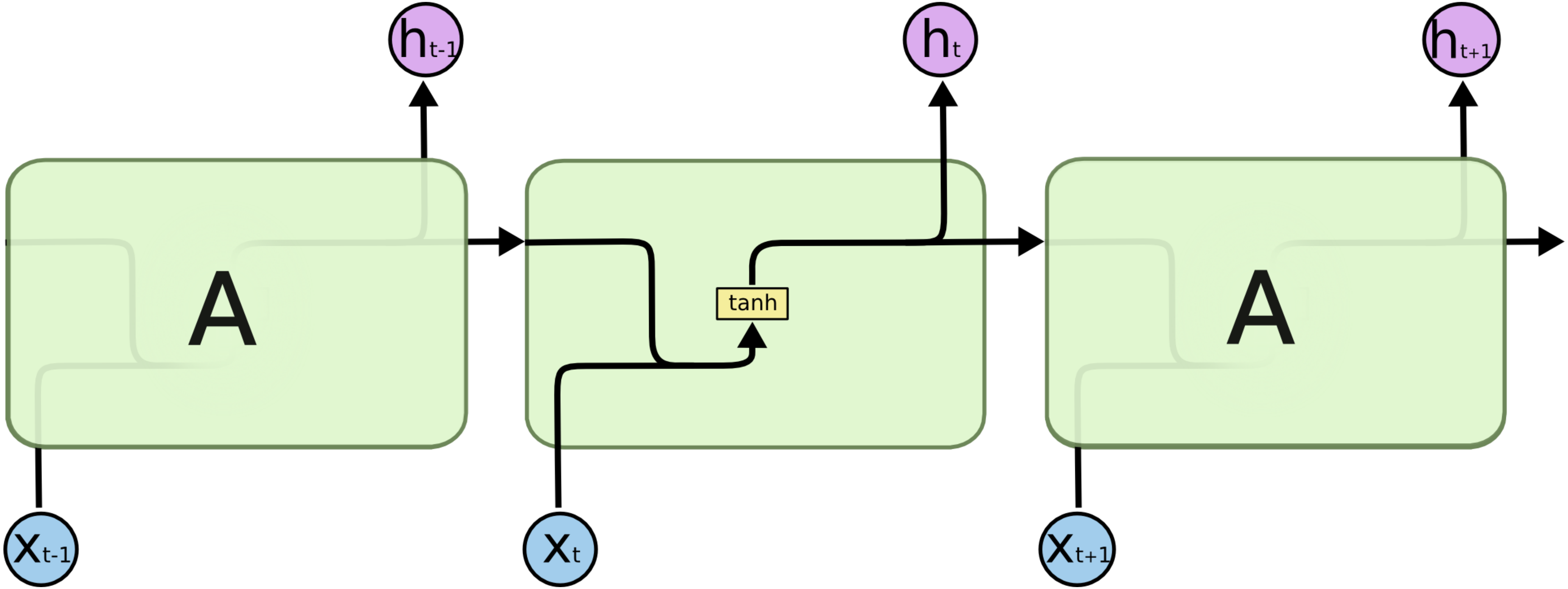
The main problem is that the *same* function is iteratively applied many times.

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

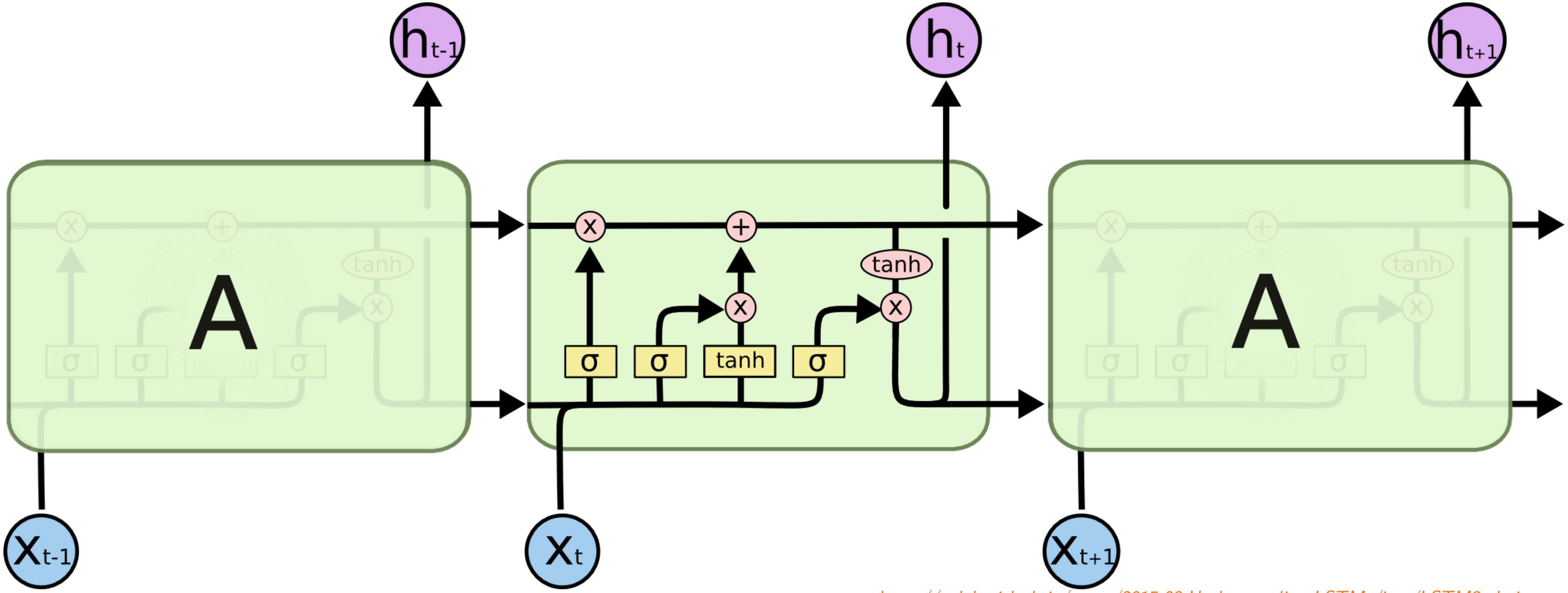
Later in Gers, Schmidhuber & Cummins (1999) a possibility to *forget* information from memory cell $\boldsymbol{c}_t$ was added.

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

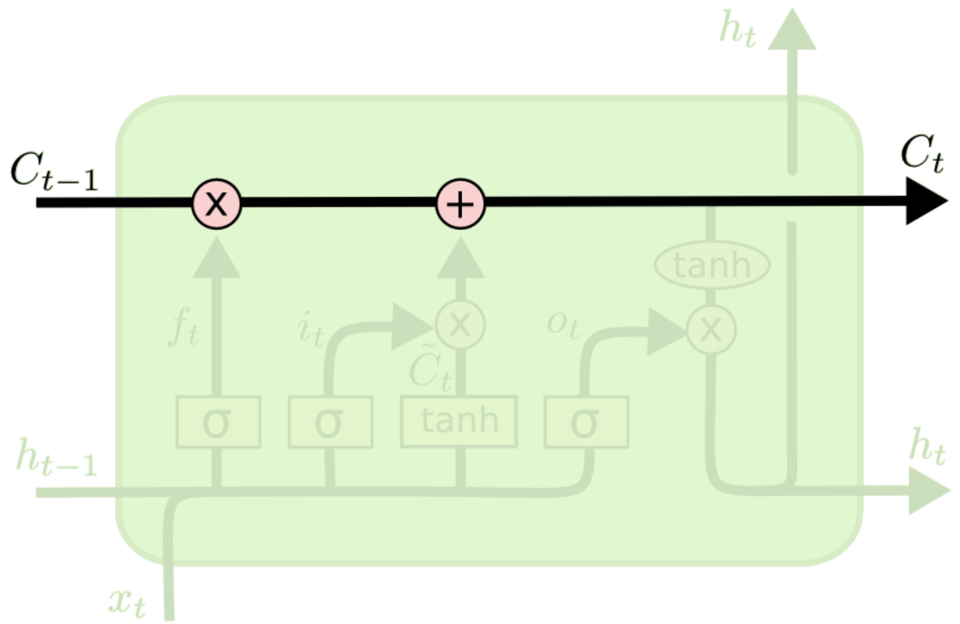




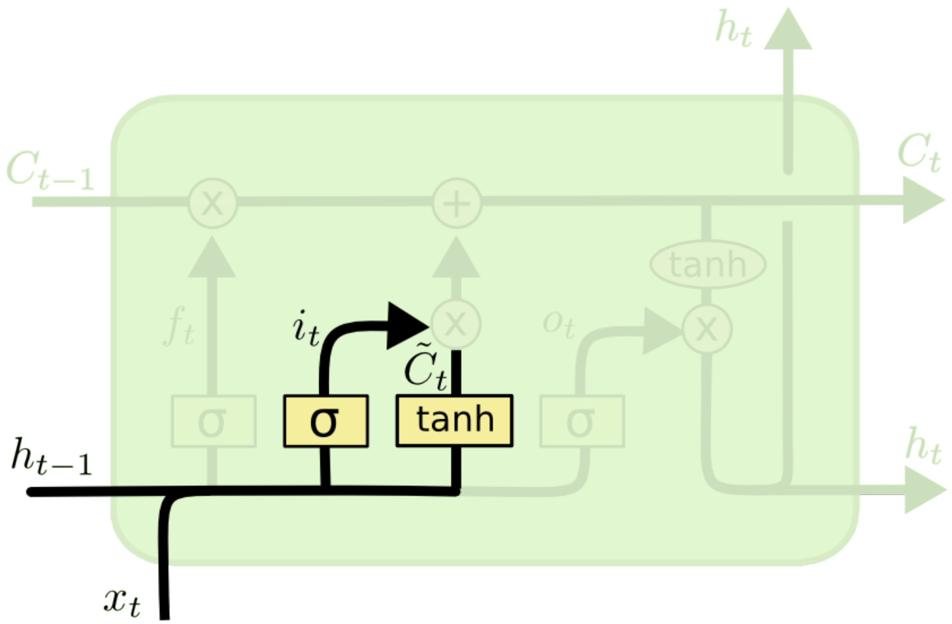
http://colah.github.io/posts/2015-08-Understanding-LSTMs/img/LSTM3-SimpleRNN.png



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

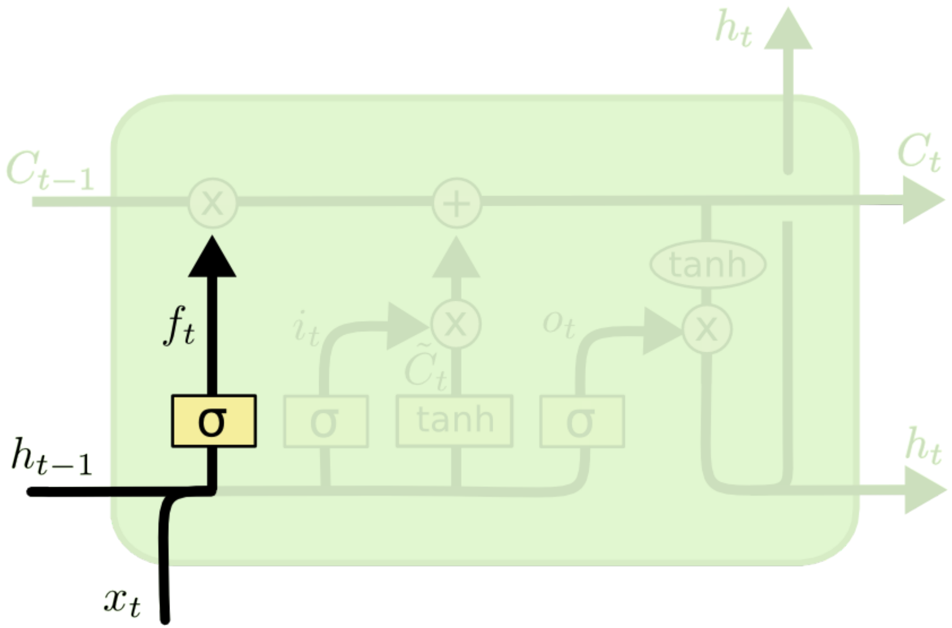


http://colah.github.io/posts/2015-08-Understanding-LSTMs/img/LSTM3-C-line.png



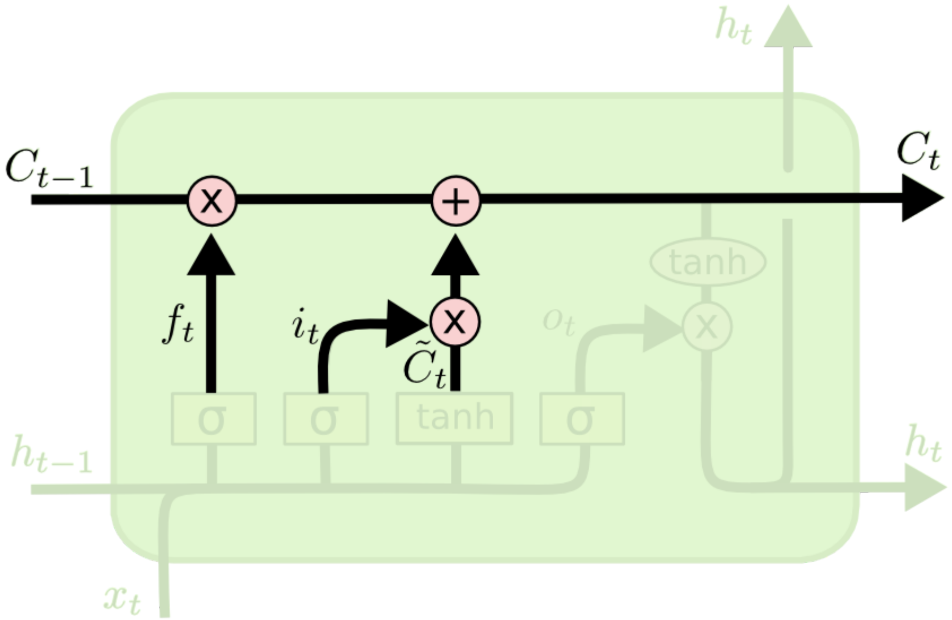
$$i_t = \sigma\left(W_i \cdot [h_{t-1}, x_t] \; + \; b_i\right)$$
$$\tilde{C}_t = \tanh(W_C \cdot [h_{t-1}, x_t] \; + \; b_C)$$

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



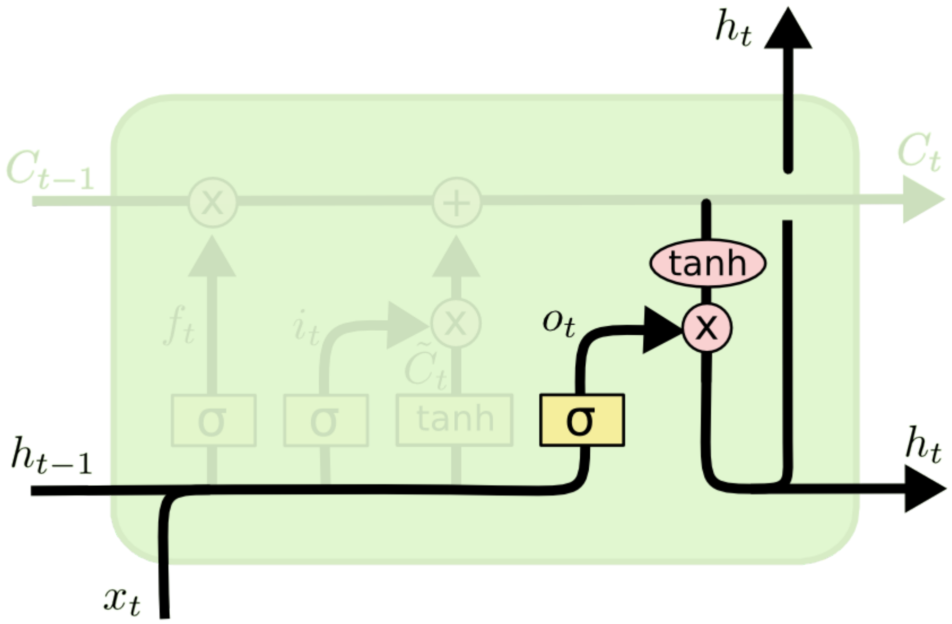
$$f_t = \sigma\left(W_f \cdot [h_{t-1}, x_t] \ + \ b_f\right)$$

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



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

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



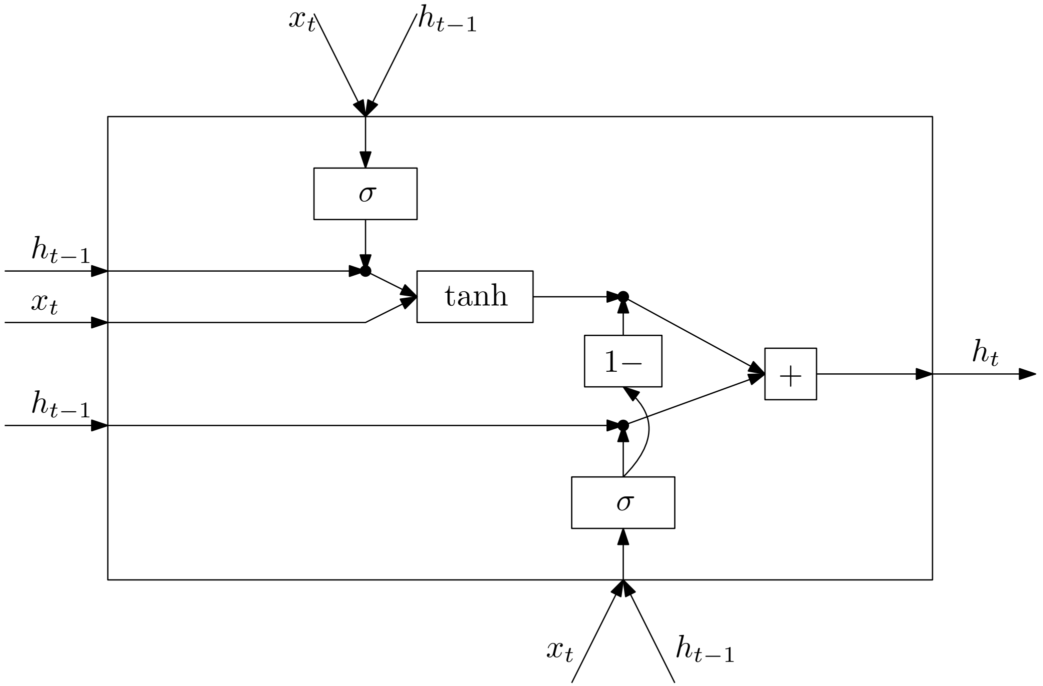
$$o_t = \sigma\left(W_o\ [h_{t-1}, x_t]\ +\ b_o\right)$$
$$h_t = o_t * \tanh\left(C_t\right)$$

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

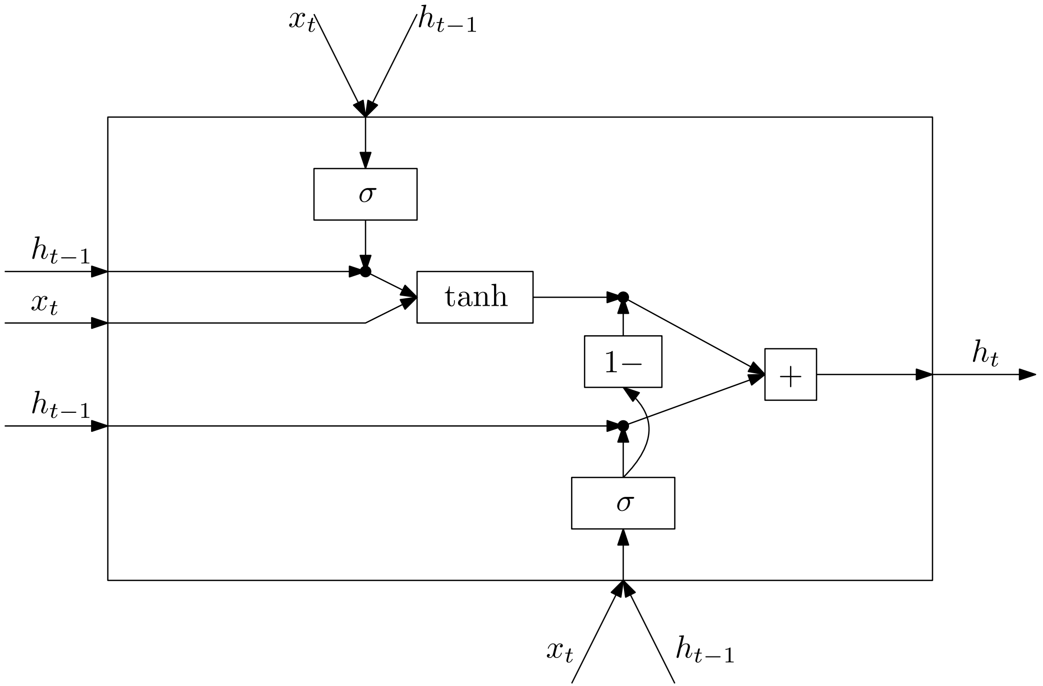
Gated Recurrent Unit

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



Gated Recurrent Unit

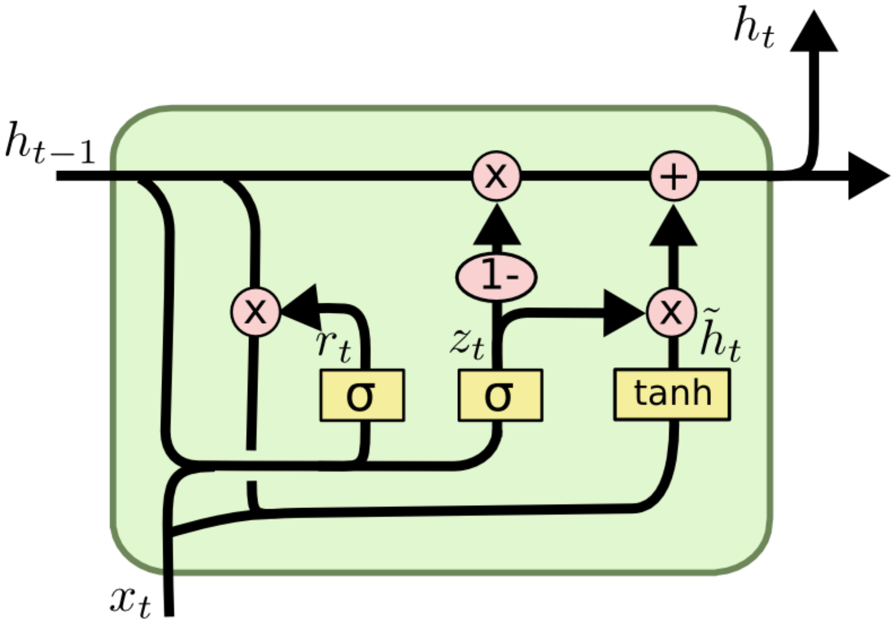


$$\boldsymbol r_t \leftarrow \sigma(\boldsymbol W^r \boldsymbol x_t + \boldsymbol V^r \boldsymbol h_{t-1} + \boldsymbol b^r)$$

$$\boldsymbol u_t \leftarrow \sigma(\boldsymbol W^u \boldsymbol x_t + \boldsymbol V^u \boldsymbol h_{t-1} + \boldsymbol b^u)$$

$$\hat{\boldsymbol h}_t \leftarrow \tanh(\boldsymbol W^h \boldsymbol x_t + \boldsymbol V^h (\boldsymbol r_t \cdot \boldsymbol h_{t-1}) + \boldsymbol b^h)$$

$$\boldsymbol h_t \leftarrow \boldsymbol u_t \cdot \boldsymbol h_{t-1} + (1 - \boldsymbol u_t) \cdot \hat{\boldsymbol h}_t$$



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

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

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

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

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

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

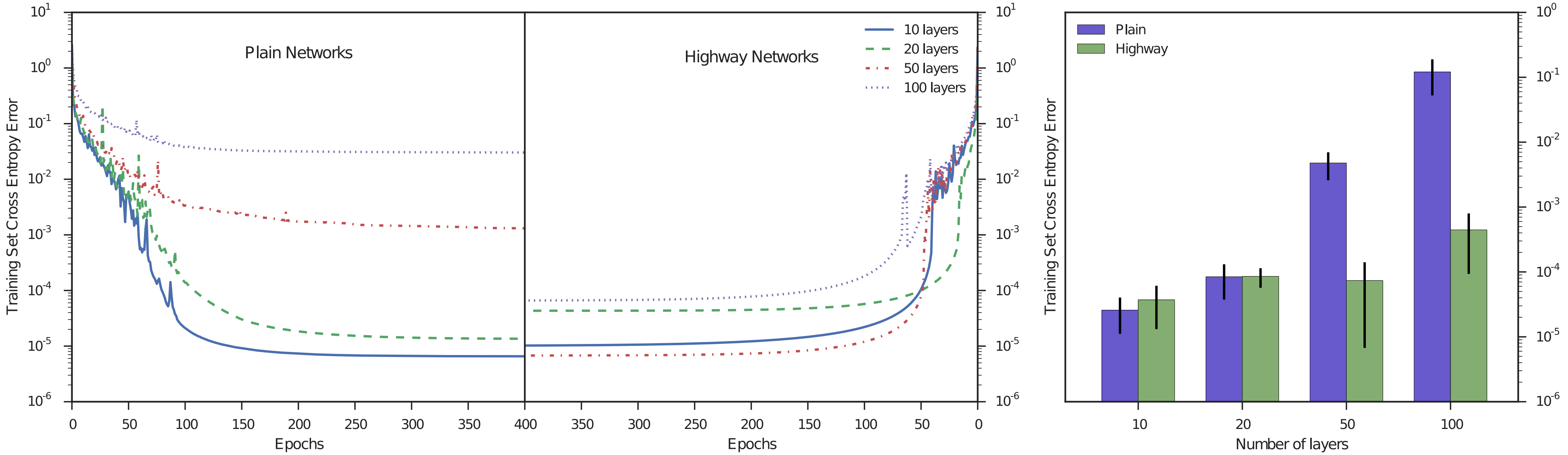


Figure 1: Comparison of optimization of plain networks and highway networks of various depths. *Left:* The training curves for the best hyperparameter settings obtained for each network depth. *Right:* Mean performance of top 10 (out of 100) hyperparameter settings. Plain n tworks become much harder 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

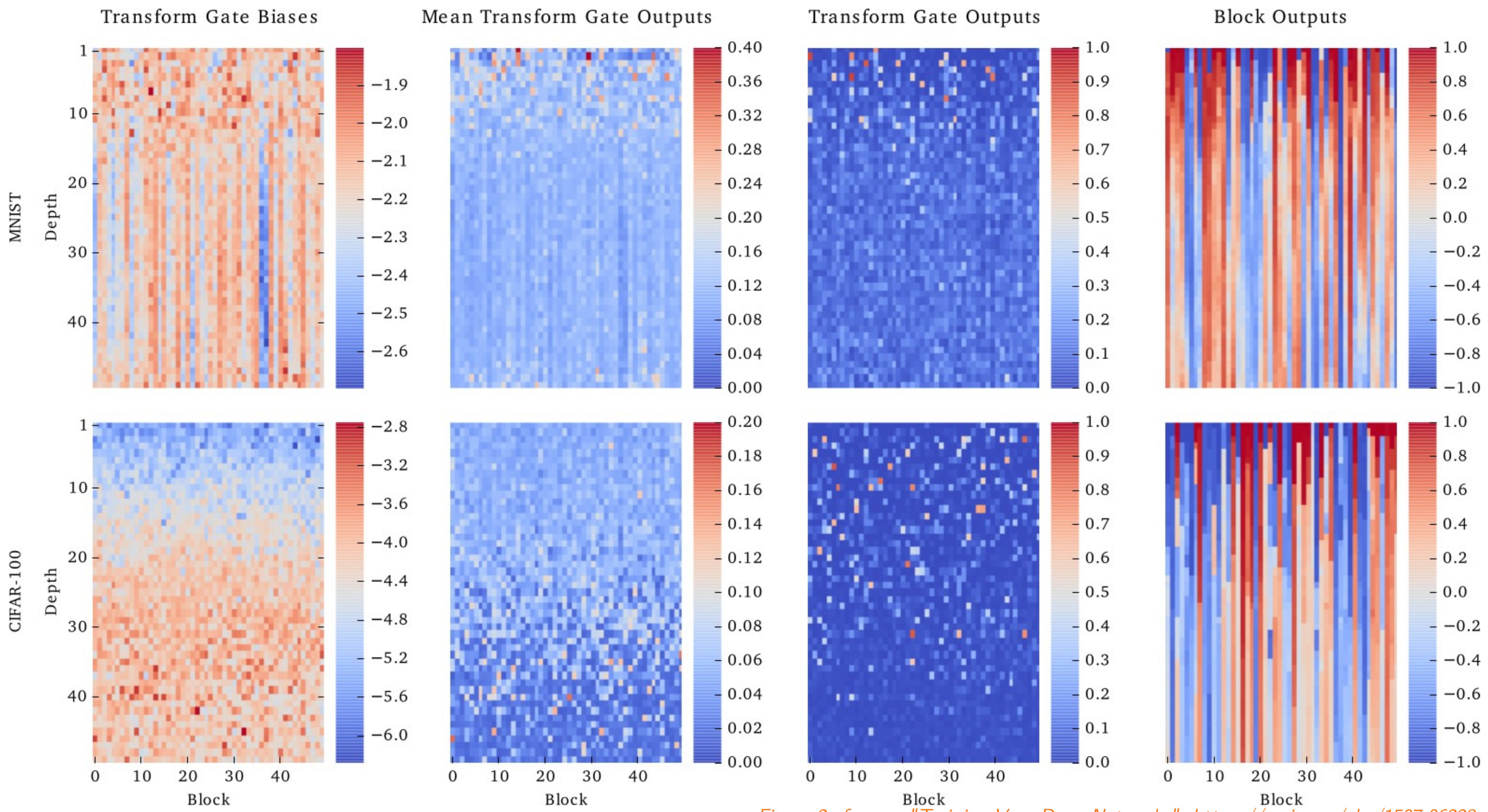


Figure 2 of paper "Training Very Deep Networks", https://arxiv.org/abs/1507.06228.

Highway Networks

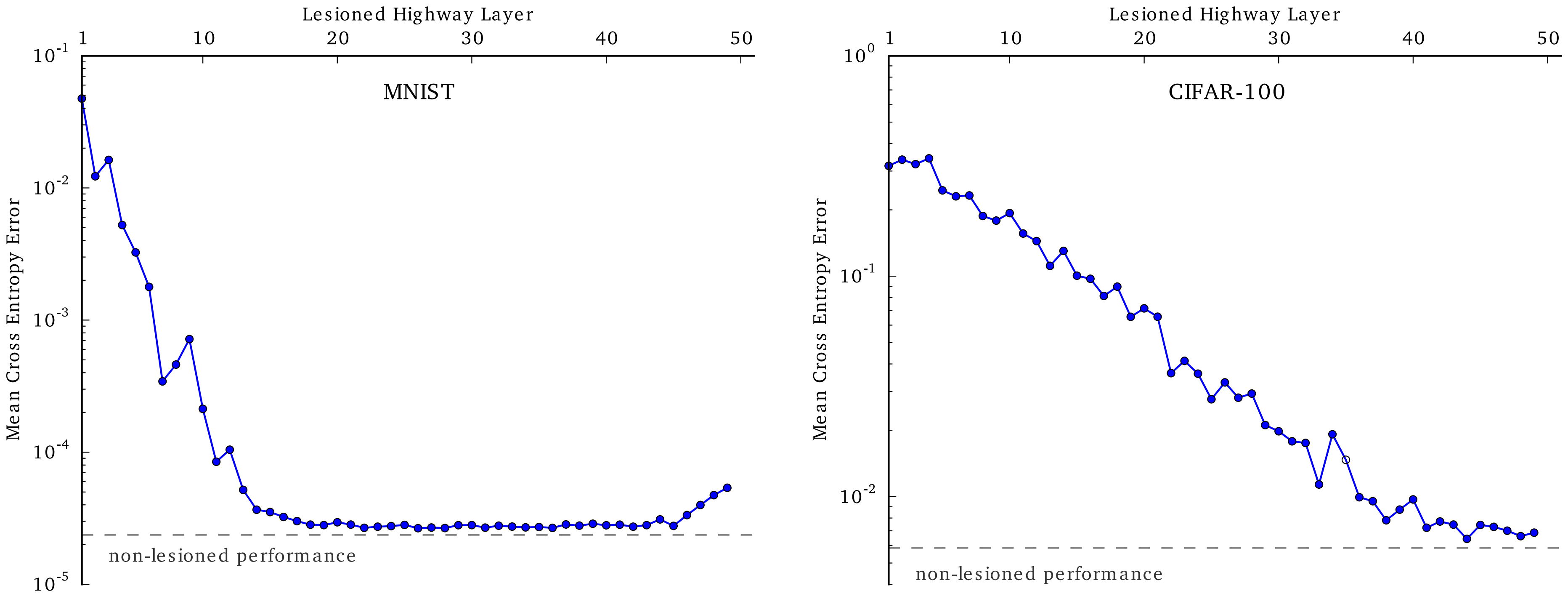
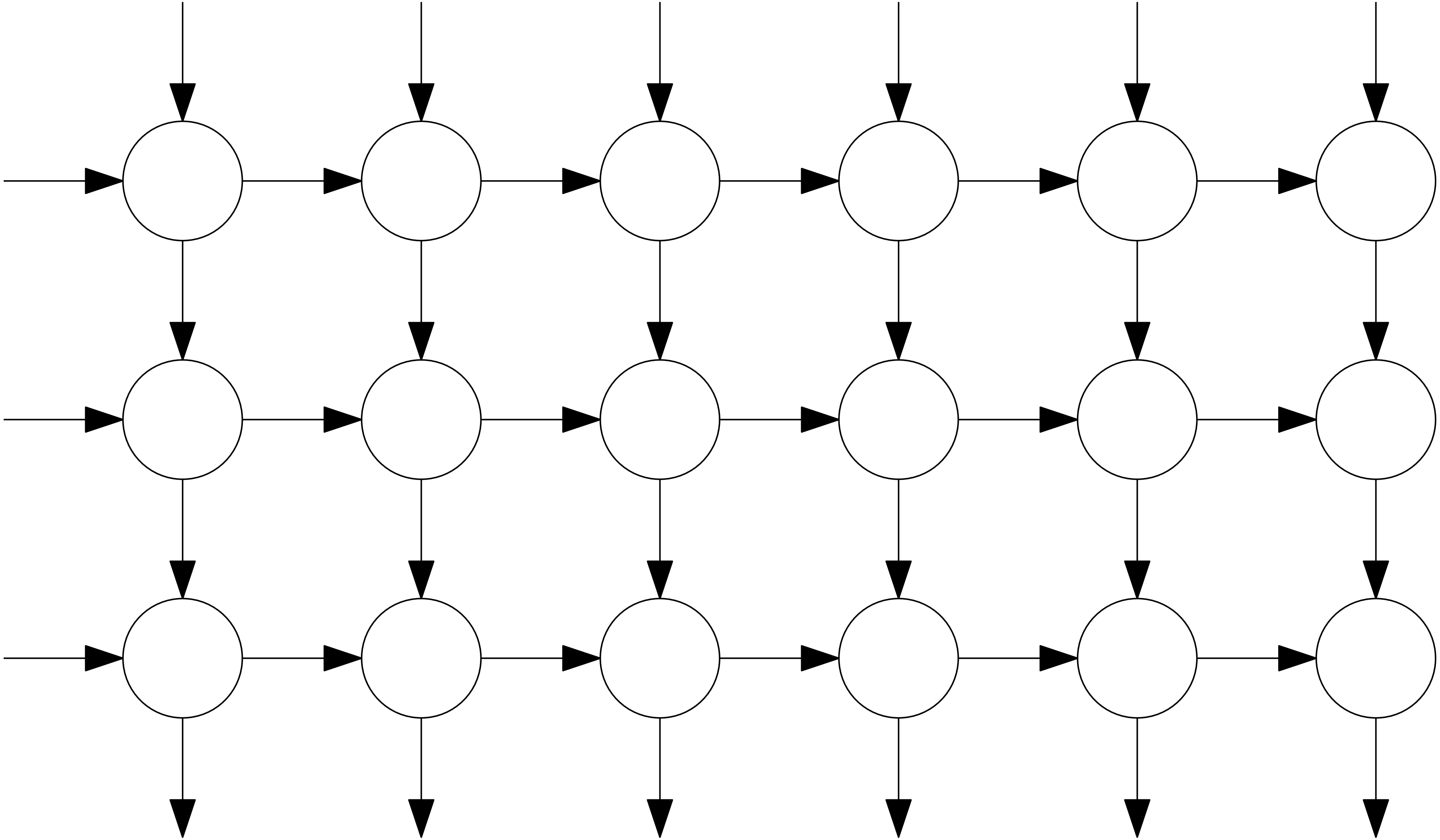
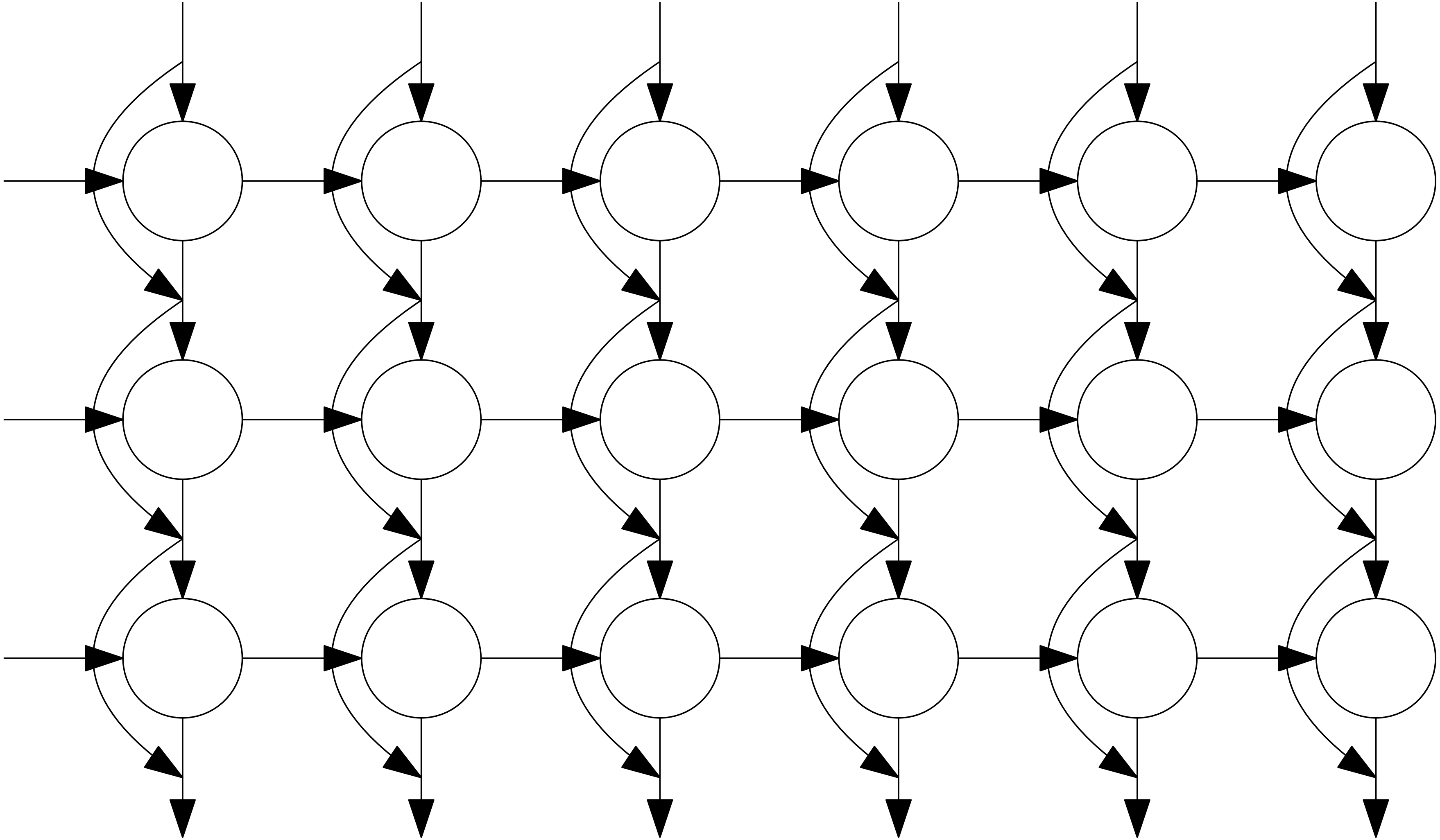


Figure 4: Lesioned training set performance (y-axis) of he best 50-layer highway networks on MNIST (left) and CIFAR-100 (right), as a function of the lesioned layer (x-axis). Evaluated on the full training set while forcefully closing all the transform gates of a single layer at a time. The non-lesioned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

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



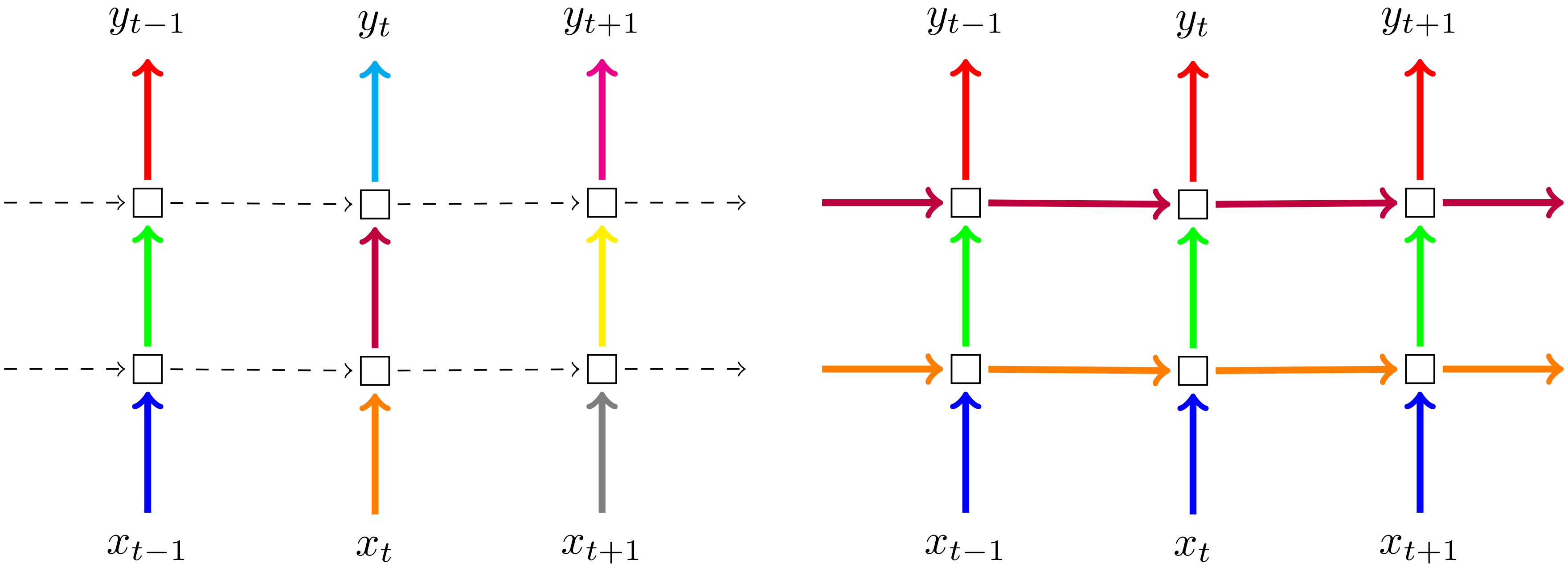
Multilayer RNNs



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

Implemented in `tf.keras.layers.{RNN,LSTM,GRU}` using `dropout` and `recurrent_dropout` arguments (for dropping inputs and previous states, respectively).

Regularizing RNNs

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 until people managed to make it work).

Layer Normalization

Much more stable than batch normalization.

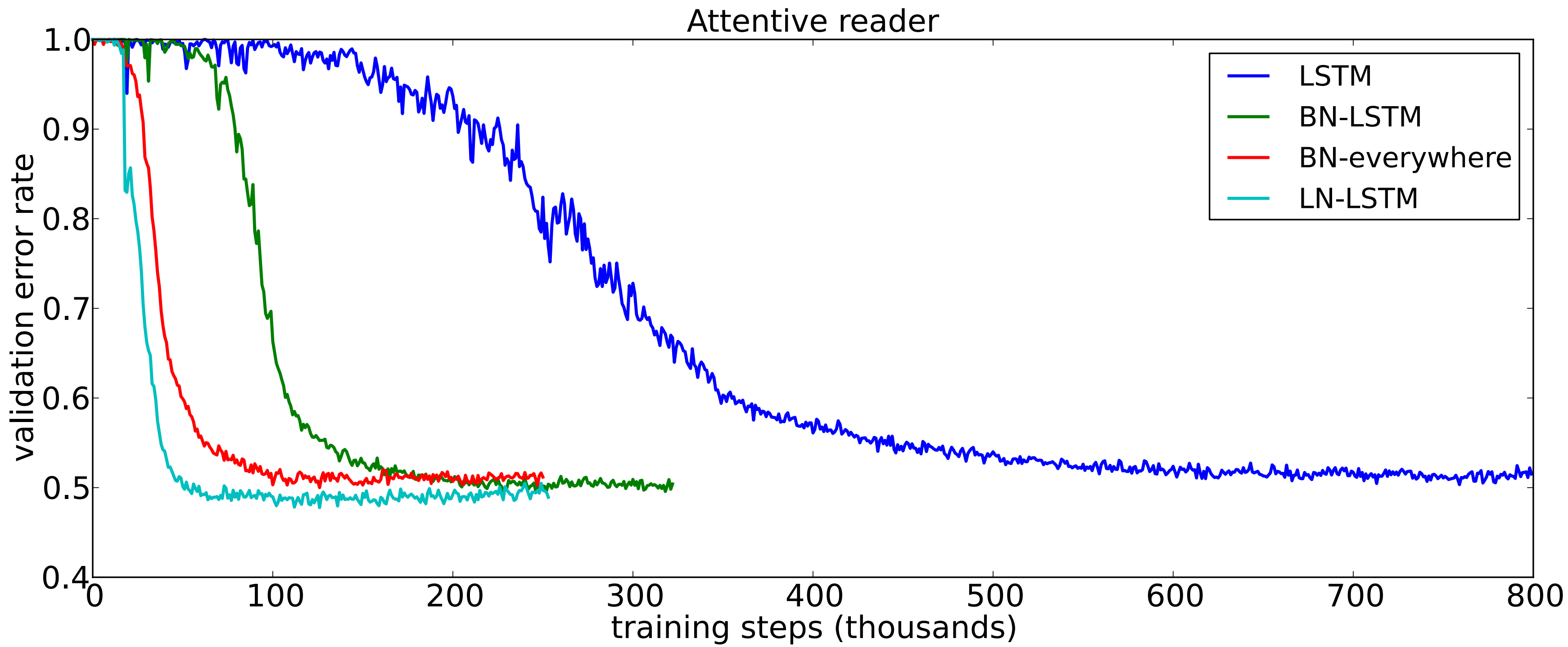


Figure 2: Validation curves for the attentive reader model. BN results are taken from [Cooijmans et al., 2016].

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

One-hot encoding considers 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.

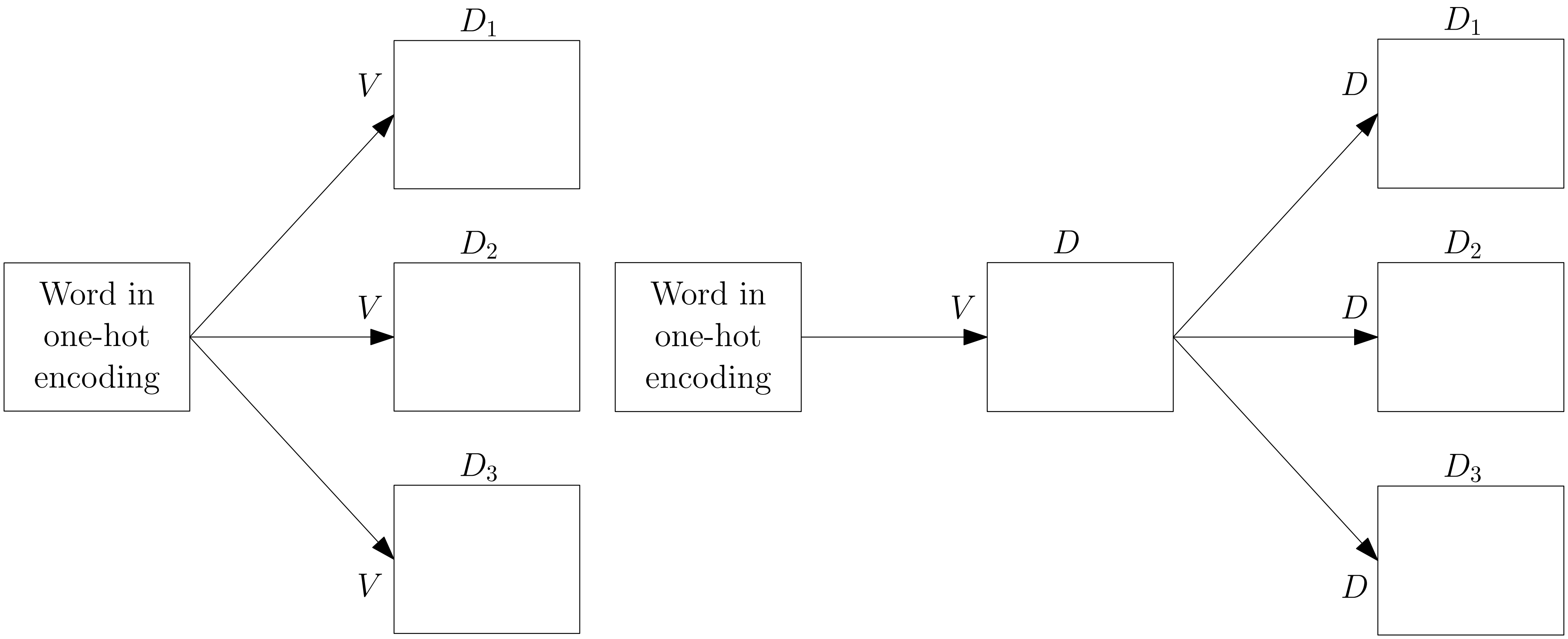
Distributed Representation

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.

Word Embeddings

The word embedding layer is in fact just a fully connected layer on top of one-hot encoding. However, it is important that this layer is *shared* across the whole network.



Recurrent Character-level WEs

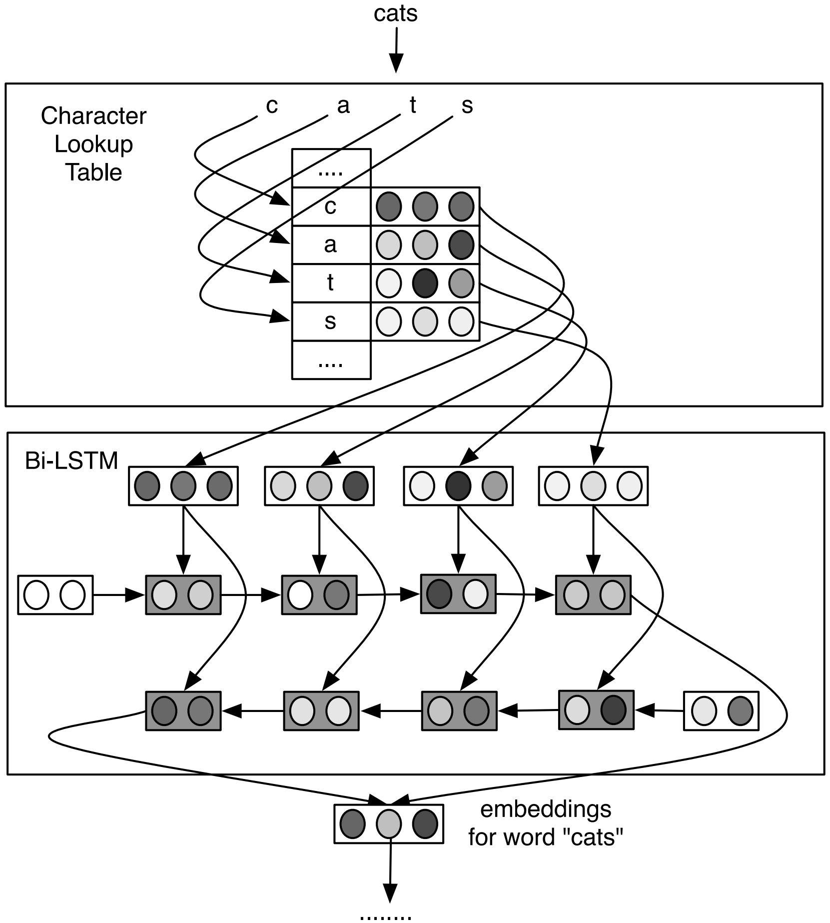


Figure 1 of paper "Finding Function in Form: Compositional Character Models for Open Vocabulary Word Representation", https://arxiv.org/abs/1508.02096.

Convolutional Character-level WEs

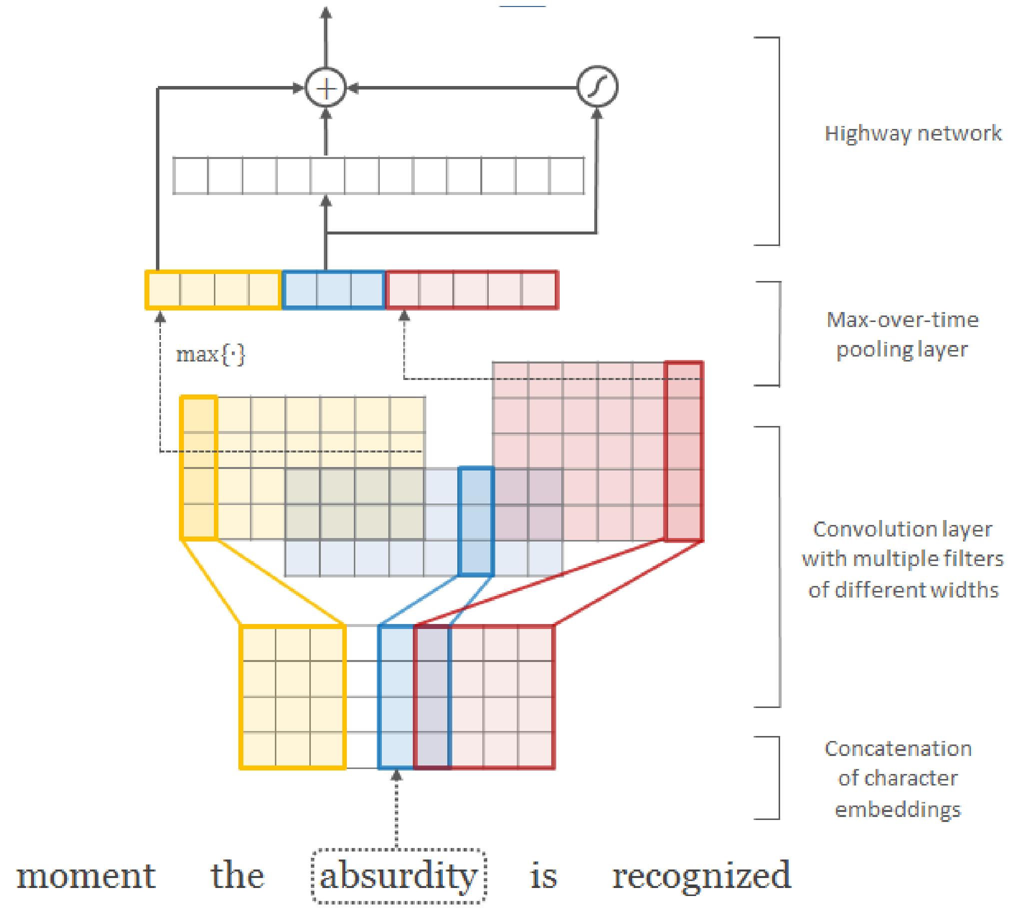


Figure 1 of paper "Character-Aware Neural Language Models", https://arxiv.org/abs/1508.06615.

Character-level WE Implementation

Training

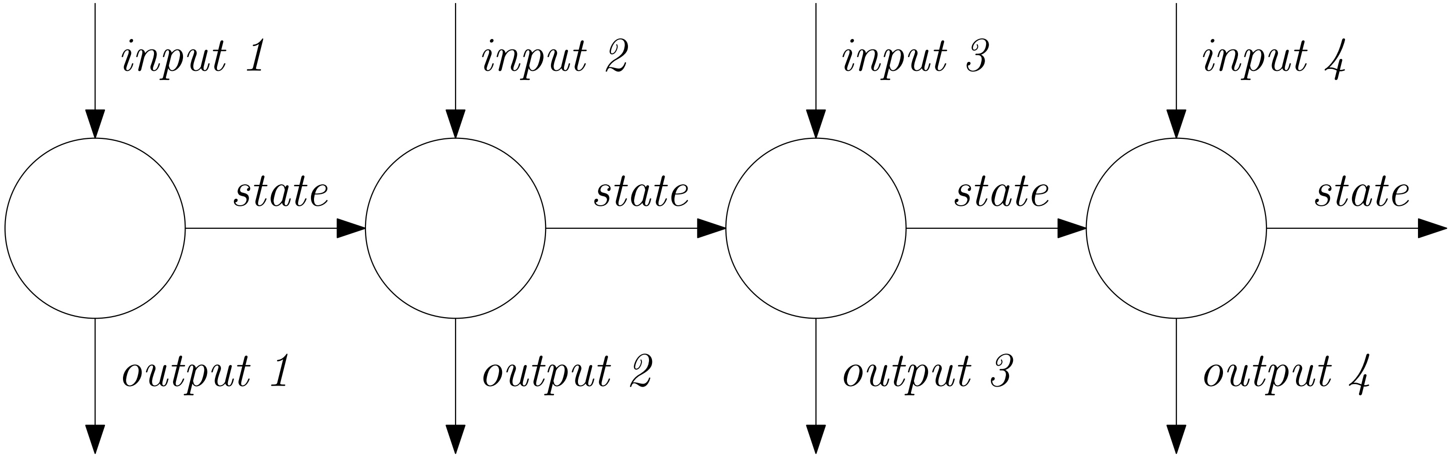
- Generate unique words per batch.
- Process the unique words in the batch.
- Copy the resulting embeddings suitably in the batch.

Inference

- We can cache character-level word embeddings during inference.

Sequence Element Classification

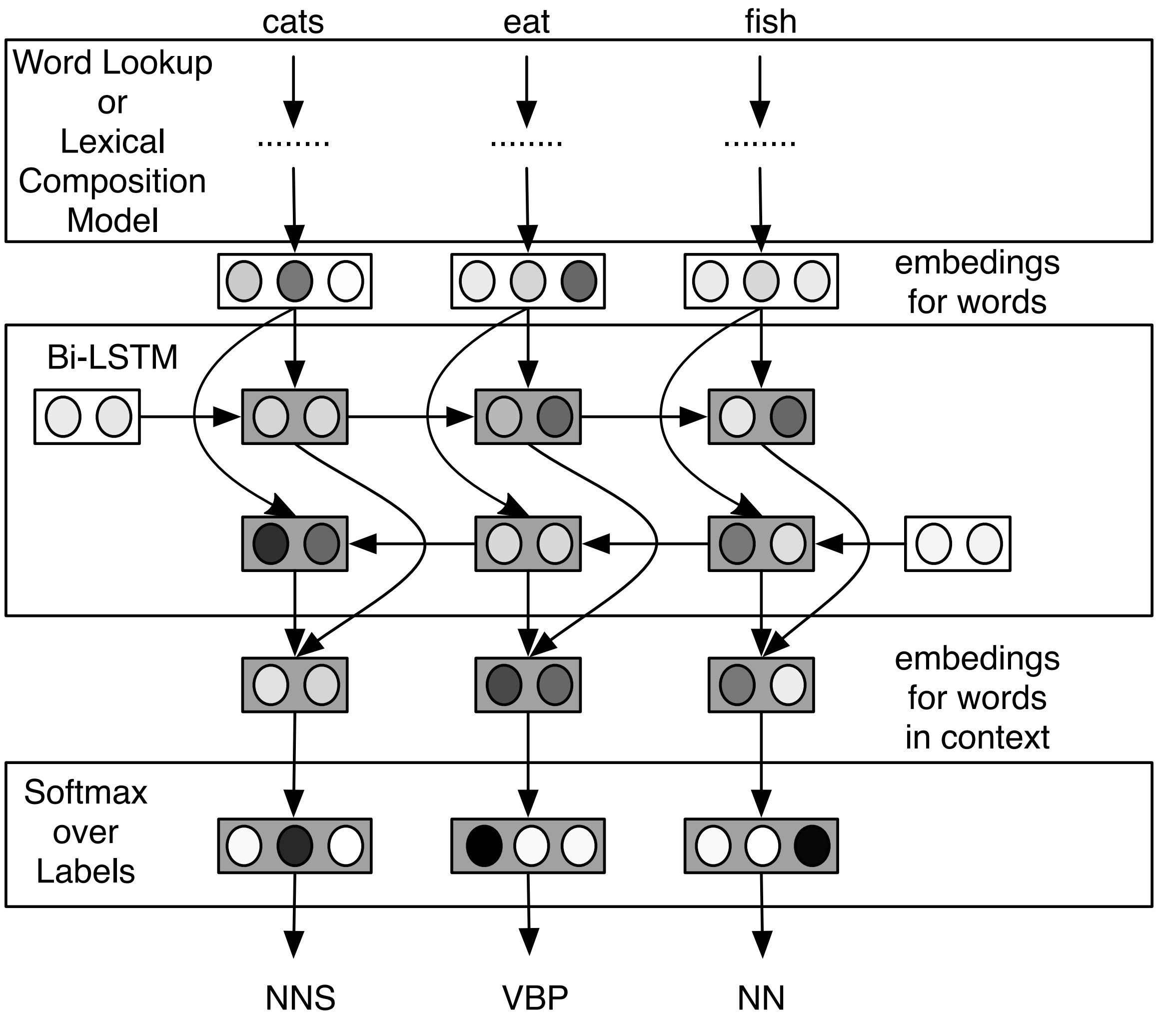
Use outputs for individual elements.



Sequence Representation

Use state after processing the whole sequence (alternatively, take output of the last element).

Bidirectional RNN



Modified Figure 3 of paper "Finding Function in Form: Compositional Character Models for Open Vocabulary Word Representation", https://arxiv.org/abs/1508.02096.

Sequence Tagging

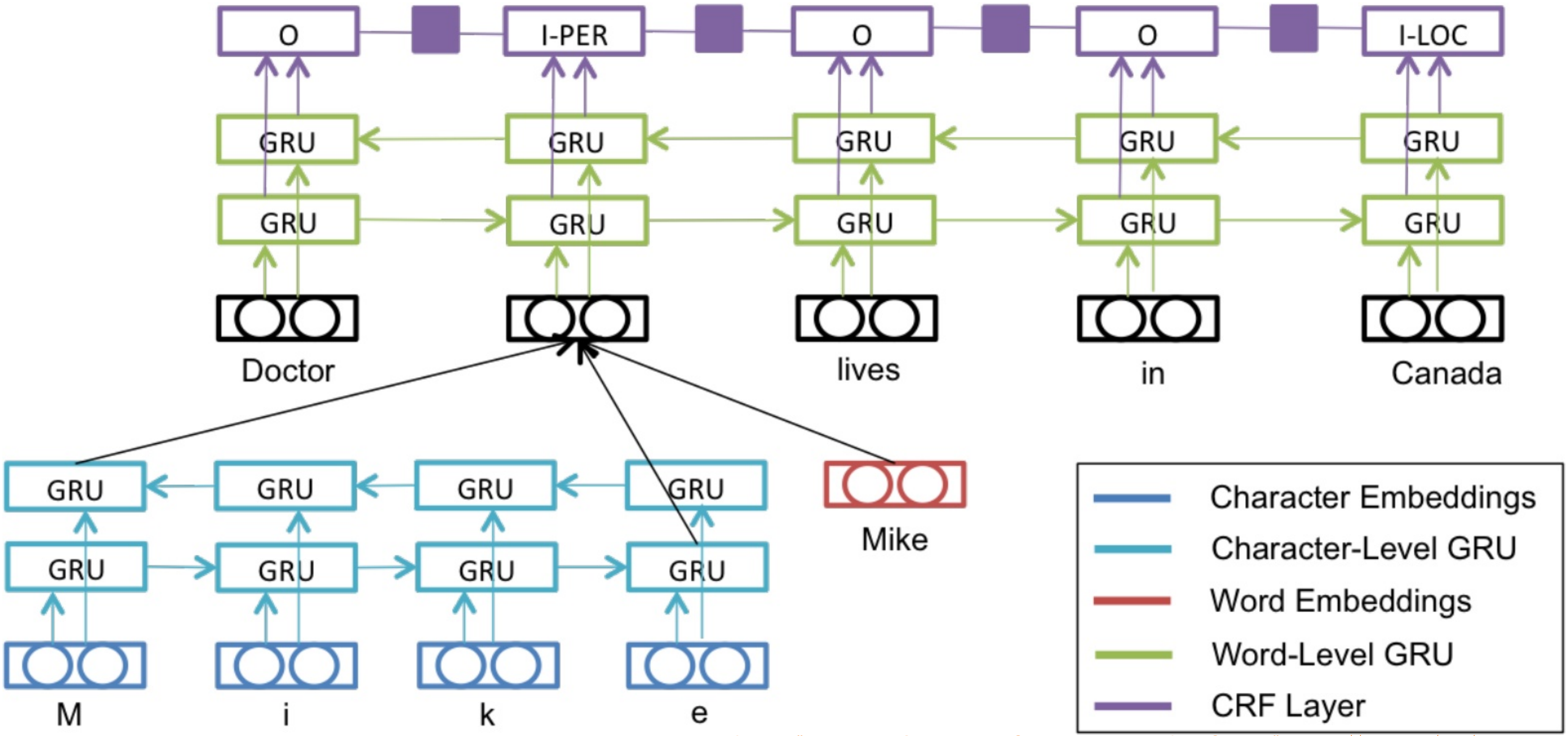
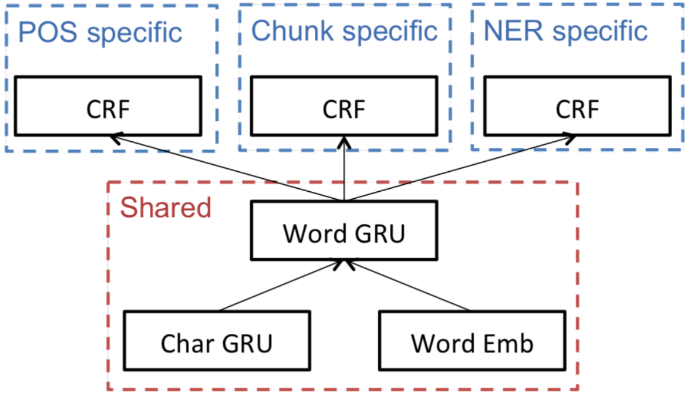
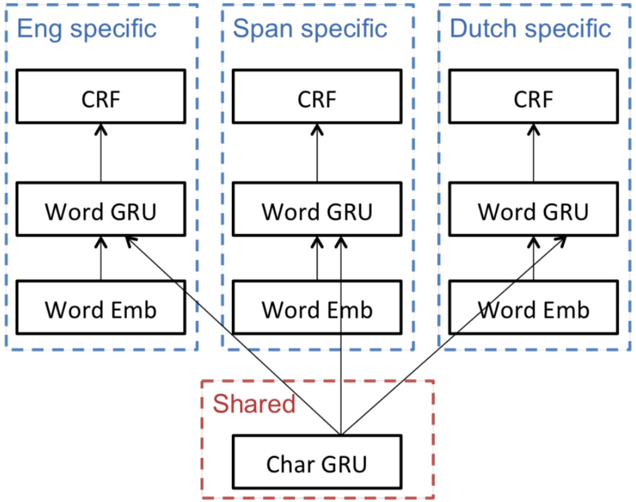


Figure 1 of paper "Multi-Task Cross-Lingual Sequence Tagging from Scratch", https://arxiv.org/abs/1603.06270.

Multitask Learning



(a) Multi-Task Joint Training



(b) Cross-Lingual Joint Training

Figure 2 of paper "Multi-Task Cross-Lingual Sequence Tagging from Scratch", https://arxiv.org/abs/1603.06270.

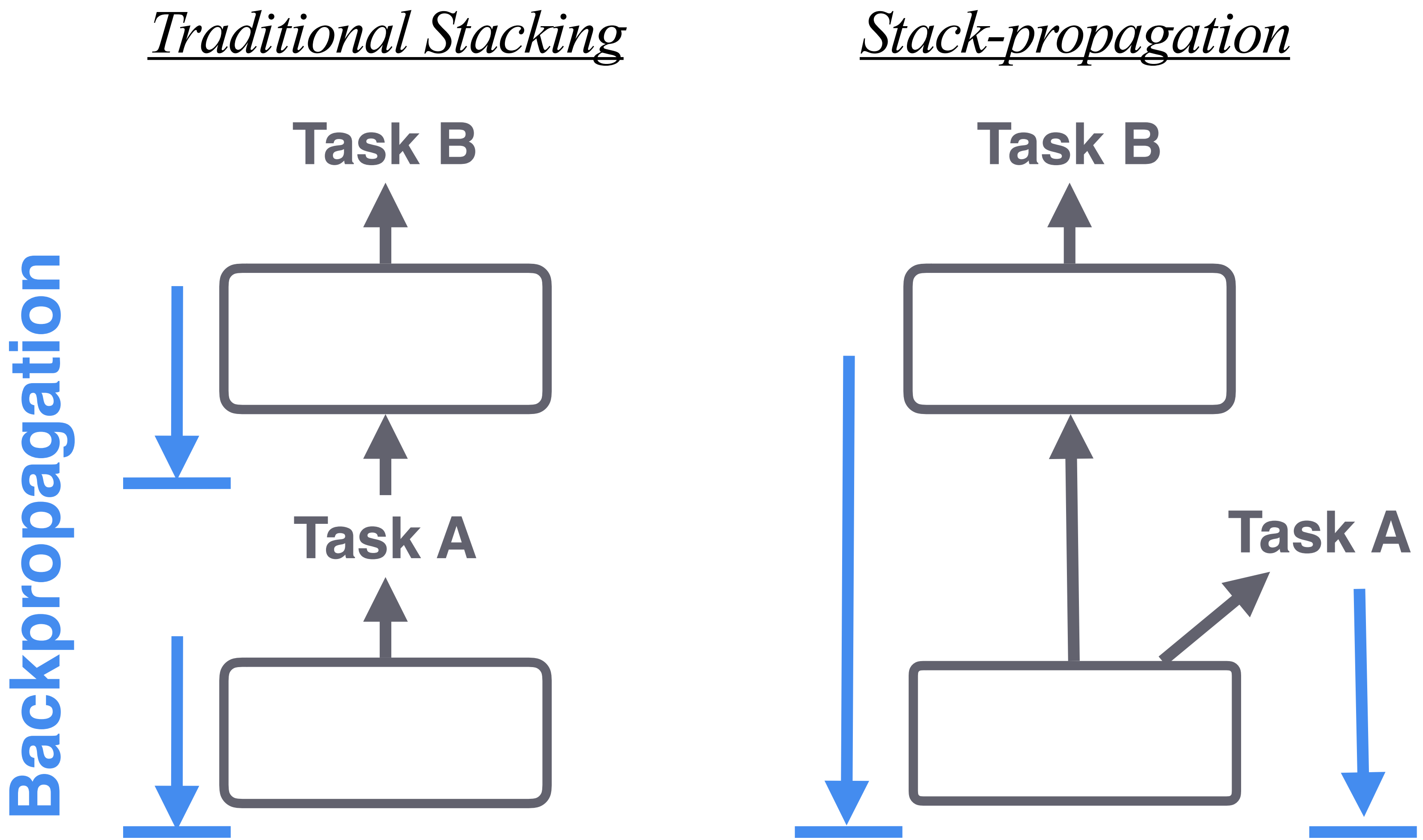


Figure 1 of paper "Stack-propagation: Improved Representation Learning for Syntax", https://arxiv.org/abs/1603.06598.

Multitask Learning

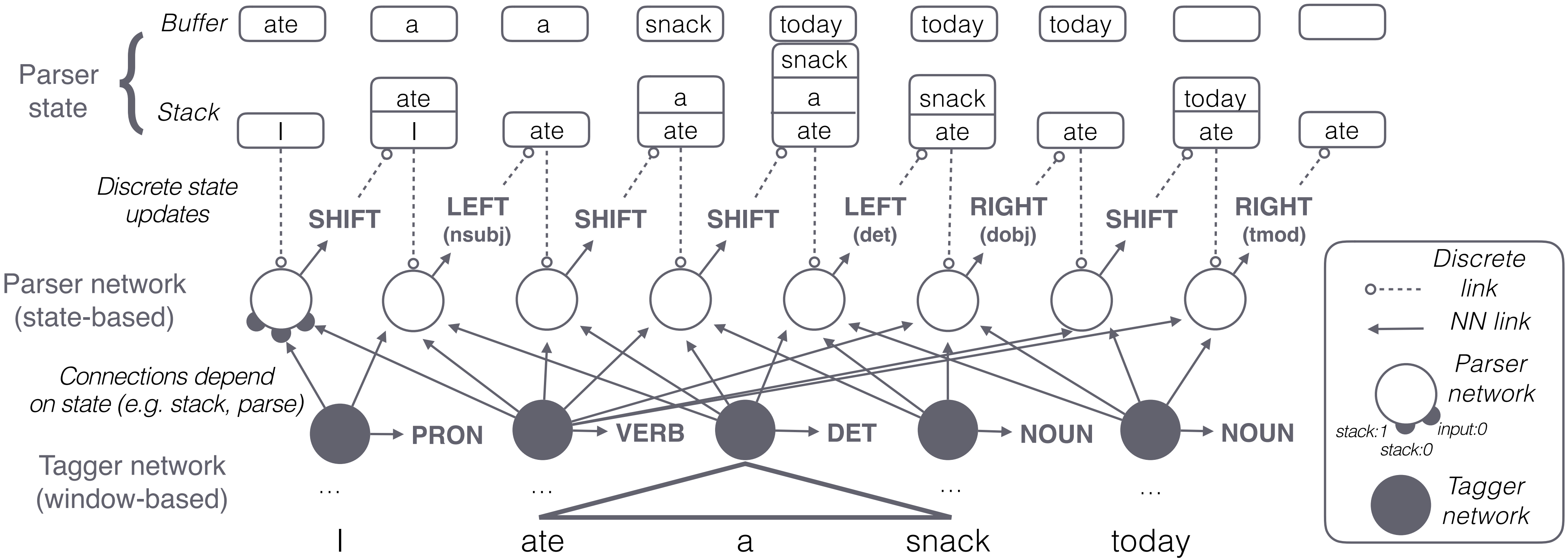


Figure 2 of paper "Stack-propagation: Improved Representation Learning for Syntax", https://arxiv.org/abs/1603.06598.

Multitask Learning

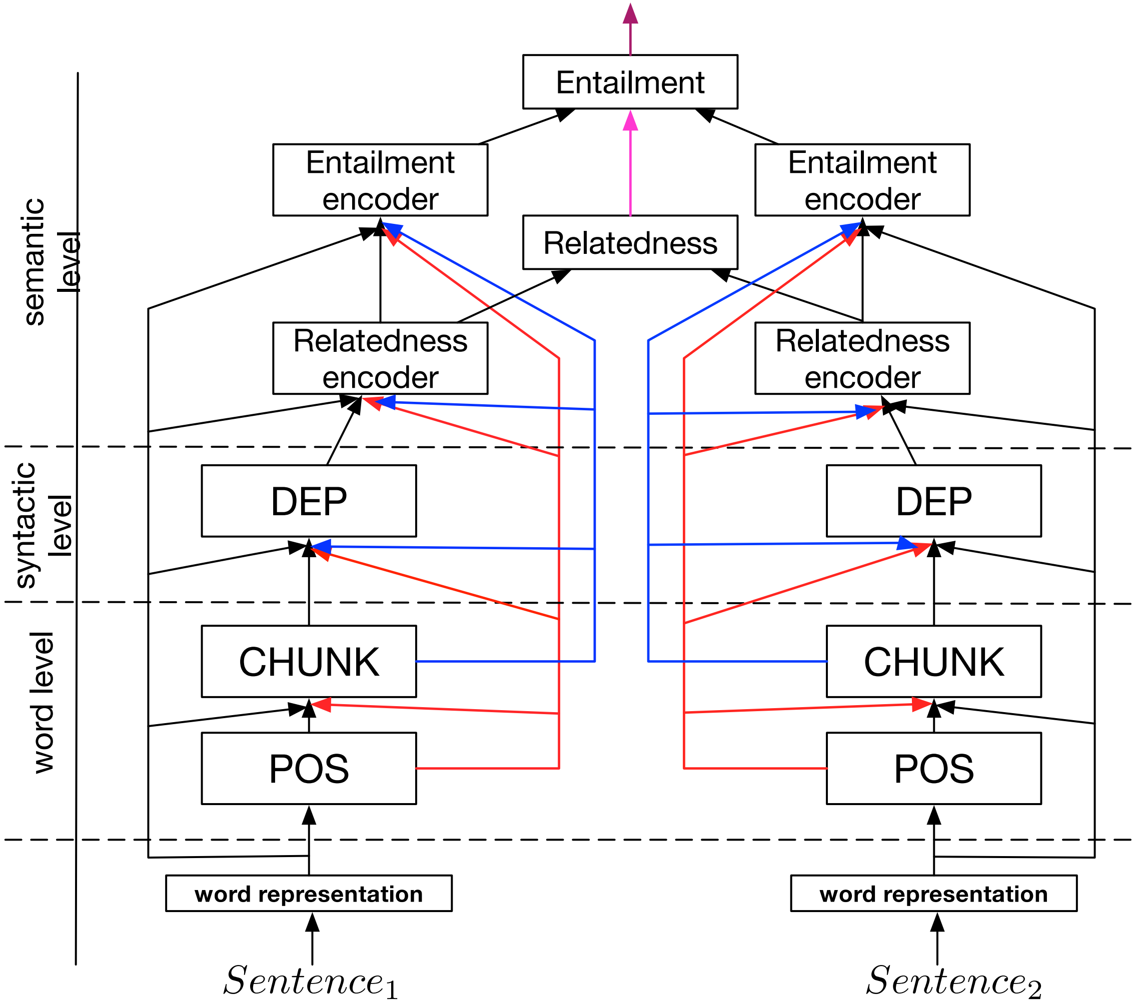


Figure 1 of paper "A Joint Many-Task Model: Growing a Neural Network for Multiple NLP Tasks", https://arxiv.org/abs/1611.01587.

Structured Prediction

Structured Prediction

Consider generating a sequence of $y_1, \ldots, y_N \in Y^N$ given input $\boldsymbol{x}_1, \ldots, \boldsymbol{x}_N$.

Predicting each sequence element independently models the distribution $P(y_i | \boldsymbol{X})$.

However, there may be dependencies among the y_i themselves, which is difficult to capture by independent element classification.

Conditional Random Fields

Let $G = (V, E)$ be a graph such that Y is indexed by vertices of G. Then $(\boldsymbol{X}, \boldsymbol{y})$ is a conditional Markov field, if the random variables $\boldsymbol{y}$ conditioned on $\boldsymbol{X}$ obey the Markov property with respect to the graph, i.e.,

$$P(y_i | \boldsymbol{X}, y_j, i \neq j) = P(y_i | \boldsymbol{X}, y_j \; \forall j : (i, j) \in E).$$

Usually we assume that dependencies of $\boldsymbol{y}$, conditioned on $\boldsymbol{X}$, form a chain.

Linear-Chain Conditional Random Fields (CRF)

Linear-chain Conditional Random Fields, usually abbreviated only to CRF, acts as an output layer. It can be considered an extension of a softmax – instead of a sequence of independent softmaxes, CRF is a sentence-level softmax, with additional weights for neighboring sequence elements.

$$s(\boldsymbol{X}, \boldsymbol{y}; \boldsymbol{\theta}, \boldsymbol{A}) = \sum_{i=1}^{N} \big(\boldsymbol{A}_{y_{i-1}, y_i} + f_{\boldsymbol{\theta}}(y_i | \boldsymbol{X})\big)$$

$$p(\boldsymbol{y} | \boldsymbol{X}) = \operatorname{softmax}_{\boldsymbol{z} \in Y^N} \big(s(\boldsymbol{X}, \boldsymbol{z})\big)_{\boldsymbol{z}}$$

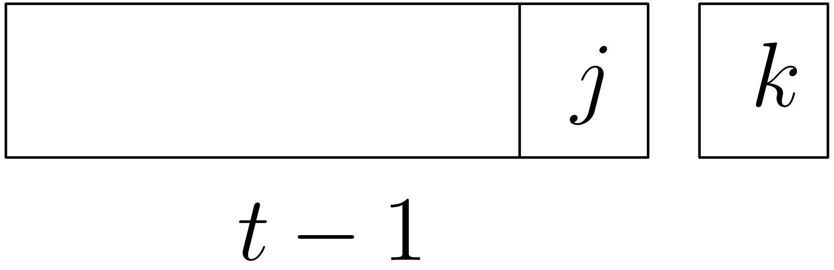
$$\log p(\boldsymbol{y} | \boldsymbol{X}) = s(\boldsymbol{X}, \boldsymbol{y}) - \operatorname{logadd}_{\boldsymbol{z} \in Y^N}(s(\boldsymbol{X}, \boldsymbol{z}))$$

Linear-Chain Conditional Random Fields (CRF)

Computation

We can compute $p(\boldsymbol{y}|\boldsymbol{X})$ efficiently using dynamic programming. We denote $\alpha_t(k)$ the logarithmic probability of all t-element sequences with the last label y being k.

The core idea is the following:



$$\alpha_t(k) = f_{\boldsymbol{\theta}}(y_t = k|\boldsymbol{X}) + \operatorname{logadd}_{j \in Y}(\alpha_{t-1}(j) + \boldsymbol{A}_{j,k}).$$

For efficient implementation, we use the fact that

$$\ln(a+b) = \ln a + \ln(1 + e^{\ln b - \ln a}).$$

Conditional Random Fields (CRF)

Inputs: Network computing $f_{\boldsymbol\theta}(y_t = k|\boldsymbol X)$, an unnormalized probability of output sequence element probability being k at time t.
Inputs: Transition matrix $\boldsymbol A \in \mathbb{R}^{Y\times Y}$.
Inputs: Input sequence $\boldsymbol X$ of length N, gold labeling $\boldsymbol y^g \in Y^N$.
Outputs: Value of $\log p(\boldsymbol y|\boldsymbol X)$.
Time Complexity: $\mathcal{O}(N \cdot Y^2)$.

- For $t = 1, \ldots, N$:
  - For $k = 1, \ldots, Y$:
    - $\alpha_t(k) \leftarrow f_{\boldsymbol\theta}(y_t = k|\boldsymbol X)$
    - If $t > 1$:
      - For $j = 1, \ldots, Y$:
        - $\alpha_t(k) \leftarrow \operatorname{logadd}(\alpha_t(k), \alpha_{t-1}(j) + \boldsymbol A_{j,k})$
- Return $\sum_{t=1}^N f_{\boldsymbol\theta}(y_t = y_t^g|\boldsymbol X) + \sum_{t=2}^N \boldsymbol A_{y_{t-1}^g, y_t^g} - \operatorname{logadd}_{k=1}^Y(\alpha_N(k))$

Decoding

We can perform optimal decoding, by using the same algorithm, only replacing logadd with $\max$ and tracking where the maximum was attained.

Applications

CRF output layers are useful for *span labeling* tasks, like

- named entity recognition
- dialog slot filling